

VISUAL***DSP++***[®] **5.0** **C/C++ Compiler Manual** **for SHARC[®] Processors**

Revision 1.2, March 2009

Part Number
82-001963-02

Analog Devices, Inc.
One Technology Way
Norwood, Mass. 02062-9106



Copyright Information

©2009 Analog Devices, Inc., ALL RIGHTS RESERVED. This document may not be reproduced in any form without prior, express written consent from Analog Devices, Inc.

Printed in the USA.

Disclaimer

Analog Devices, Inc. reserves the right to change this product without prior notice. Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices, Inc.

Trademark and Service Mark Notice

The Analog Devices logo, the CROSSCORE logo, VisualDSP++, SHARC, and EZ-KIT Lite are registered trademarks of Analog Devices, Inc.

All other brand and product names are trademarks or service marks of their respective owners.

CONTENTS

PREFACE

Purpose of This Manual	xxxi
Intended Audience	xxxi
Manual Contents	xxxii
What's New in This Manual	xxxii
Technical or Customer Support	xxxii
Supported Processors	xxxiii
Product Information	xxxiii
Analog Devices Web Site	xxxiv
VisualDSP++ Online Documentation	xxxiv
Technical Library CD	xxxv
Notation Conventions	xxxv

COMPILER

C/C++ Compiler Overview	1-3
Standard Conformance	1-4
Compiler Command-Line Interface	1-7
Running the Compiler	1-8
Compiler Command-Line Switches	1-9

CONTENTS

C/C++ Compiler Switch Summaries	1-9
C/C++ Mode Selection Switch Descriptions	1-22
-c89	1-22
-c++	1-22
C/C++ Compiler Common Switch Descriptions	1-23
sourcefile	1-23
-@ filename	1-23
-A name[tokens]	1-23
-add-debug-libpaths	1-24
-aligned-stack	1-25
-alttok	1-25
-always-inline	1-26
-annotate	1-26
-annotate-loop-instr	1-26
-auto-attrs	1-27
-build-lib	1-27
-C	1-27
-c	1-27
-compatible-pm-dm	1-27
-const-read-write	1-28
-const-strings	1-28
-D macro[=definition]	1-28
-debug-types	1-29
-double-size[-32 -64]	1-29

-double-size-any	1-30
-dry	1-31
-dryrun	1-31
-E	1-31
-ED	1-31
-EE	1-31
-eh	1-32
-enum-is-int	1-33
-extra-keywords	1-33
-file-attr name[=value]	1-33
-flags-{asm compiler lib link mem} switch [,switch2 [...]]	1-34
-float-to-int	1-34
-force-circbuf	1-34
-fp-associative	1-35
-full-version	1-35
-g	1-35
-glite	1-36
-H	1-36
-HH	1-36
-h[elp]	1-36
-I directory [{, ;} directory...]	1-36
-I-	1-37
-i	1-38
-implicit-pointers	1-38

CONTENTS

-include filename	1-39
-ipa	1-39
-L directory[{: ,}directory...]	1-39
-l library	1-39
-list-workarounds	1-40
-M	1-40
-MD	1-40
-MM	1-41
-Mo filename	1-41
-Mt name	1-41
-map filename	1-41
-mem	1-41
-multiline	1-41
-never-inline	1-42
-no-aligned-stack	1-42
-no-alttok	1-42
-no-annotate	1-42
-no-annotate-loop-instr	1-43
-no-auto-attrs	1-43
-no-builtin	1-43
-no-circbuf	1-44
-no-const-strings	1-44
-no-db	1-44
-no-defs	1-44

-no-eh	1-45
-no-extra-keywords	1-45
-no-fp-associative	1-45
-no-mem	1-45
-no-multiline	1-46
-no-progress-rep-timeout	1-46
-no-sat-associative	1-46
-no-saturation	1-46
-no-shift-to-add	1-47
-no-simd	1-47
-no-std-ass	1-47
-no-std-def	1-48
-no-std-inc	1-48
-no-std-lib	1-48
-no-threads	1-48
-no-workaround workaround_id[,workaround_id ...]	1-48
-normal-word-code	1-49
-nwc	1-49
-O[0 1]	1-49
-Oa	1-49
-Og	1-50
-Os	1-50
-Ov num	1-50
-o filename	1-52

CONTENTS

-overlay	1-52
-overlay-clobbers clobbered-regs	1-53
-P	1-53
-PP	1-54
-path-{ asm compiler lib link } pathname	1-54
-path-install directory	1-54
-path-output directory	1-54
-path-temp directory	1-54
-pch	1-55
-pchdir directory	1-55
-pgo-session session-id	1-55
-pguide	1-56
-pplist filename	1-56
-proc processor	1-57
-progress-rep-func	1-57
-progress-rep-opt	1-58
-progress-rep-timeout	1-58
-progress-rep-timeout-secs secs	1-58
-R directory[{: ,}directory ...]	1-58
-R-	1-59
-reserve register[, register ...]	1-59
-restrict-hardware-loops maximum	1-59
-sat-associative	1-60
-S	1-60

-s	1-60
-save-temps	1-60
-section id=section_name[,id=section_name...]	1-60
-short-word-code	1-62
-show	1-62
-si-revision version	1-62
-signed-bitfield	1-62
-structs-do-not-overlap	1-63
-swc	1-63
-syntax-only	1-64
-sysdefs	1-64
-T filename	1-64
-threads	1-64
-time	1-65
-U macro	1-65
-unsigned-bitfield	1-65
-v	1-66
-verbose	1-66
-version	1-66
-W{error remark suppress warn} number[,number ...]	1-67
-Werror-limit number	1-67
-Werror-warnings	1-67
-Wremarks	1-68
-Wterse	1-68

CONTENTS

-w	1-68
-warn-protos	1-68
-workaround workaround_id[,workaround_id ...]	1-69
-write-files	1-69
-write-opts	1-69
-xref filename	1-70
C Mode (MISRA) Compiler Switch Descriptions	1-71
-misra	1-71
-misra-linkdir directory	1-71
-misra-no-cross-module	1-71
-misra-no-runtime	1-71
-misra-strict	1-72
-misra-suppress-advisory	1-72
-misra-testing	1-72
-Wmis_suppress rule_number [, rule_number]	1-72
-Wmis_warn rule_number [, rule_number]	1-73
C++ Mode Compiler Switch Descriptions	1-73
-anach	1-73
-check-init-order	1-74
-full-dependency-inclusion	1-75
-ignore-std	1-75
-no-anach	1-76
-no-implicit-inclusion	1-76
-no-rtti	1-76

-no-std-templates	1-76
-rtti	1-76
-std-templates	1-77
Environment Variables Used by the Compiler	1-77
Data Type and Data Type Sizes	1-78
Integer Data Types	1-79
Floating-Point Data Types	1-79
Optimization Control	1-80
Optimization Levels	1-81
Interprocedural Analysis	1-84
Interaction with Libraries	1-84
Controlling Silicon Revision and Anomaly Workarounds within the Compiler	1-85
Using the -si-revision Switch	1-86
Using the -workaround Switch	1-88
Using the -no-workaround Switch	1-89
Interactions Between the Silicon Revision and Workaround Switches 1-89	
MISRA-C Compiler	1-91
MISRA-C Compiler Overview	1-91
MISRA-C Compliance	1-92
Using the Compiler to Achieve Compliance	1-92
Rules Descriptions	1-95
C/C++ Compiler Language Extensions	1-104
Function Inlining	1-108

CONTENTS

Inlining and Optimization	1-111
Inlining and Out-of-Line Copies	1-111
Inlining and Global asm Statements	1-112
Inlining and Sections	1-112
Inline Assembly Language Support Keyword (asm)	1-113
asm() Construct Syntax	1-115
asm() Construct Syntax Rules	1-116
asm() Construct Template Example	1-117
Assembly Construct Operand Description	1-118
Assembly Constructs With Multiple Instructions	1-124
Assembly Construct Reordering and Optimization	1-125
Assembly Constructs With Input and Output Operands ..	1-126
Assembly Constructs With Compile-Time Constants	1-127
Assembly Constructs and Flow Control	1-128
Guidelines on the Use of asm() Statements	1-129
Dual Memory Support Keywords (pm dm)	1-130
Memory Keywords and Assignments/Type Conversions ...	1-132
Memory Keywords and Function Declarations/Pointers ...	1-133
Memory Keywords and Function Arguments	1-133
Memory Keywords and Macros	1-134
Bank Type Qualifiers	1-135
Placement Support Keyword (section)	1-136
Placement of Compiler-Generated Code and Data	1-137
Boolean Type Support Keywords (bool, true, false)	1-139

Pointer Class Support Keyword (restrict)	1-139
Variable-Length Array Support	1-140
Long Identifiers	1-141
Non-Constant Initializer Support	1-142
Indexed Initializer Support	1-142
Aggregate Constructor Expression Support	1-144
Preprocessor Generated Warnings	1-145
C++ Style Comments	1-145
Compiler Built-In Functions	1-146
Access to System Registers	1-146
Circular Buffer Built-In Functions	1-151
Circular Buffer Increment of an Index	1-151
Circular Buffer Increment of a Pointer	1-151
Compiler Performance Built-In Functions	1-152
Expected Behavior	1-152
Known Values	1-153
Fractional Built-In Functions	1-155
Miscellaneous Built-In Functions	1-156
Pragmas	1-156
Data Alignment Pragmas	1-158
#pragma align num	1-158
#pragma alignment_region (alignopt)	1-160
#pragma pack (alignopt)	1-161
#pragma pad (alignopt)	1-162

CONTENTS

Interrupt Handler Pragas	1-162
#pragma implicit_push_sts_handler	1-163
#pragma interrupt	1-163
#pragma interrupt_complete_nesting	1-164
#pragma interrupt_complete	1-165
Interrupt Pragas and the Interrupt Vector Table	1-165
Loop Optimization Pragas	1-166
#pragma SIMD_for	1-166
#pragma all_aligned	1-167
#pragma no_vectorization	1-167
#pragma loop_count (min, max, modulo)	1-167
#pragma loop_unroll N	1-168
#pragma no_alias	1-170
#pragma vector_for	1-171
General Optimization Pragas	1-171
Function Side-Effect Pragas	1-173
#pragma alloc	1-173
#pragma const	1-174
#pragma misra_func(arg)	1-174
#pragma noreturn	1-174
#pragma pgo_ignore	1-175
#pragma pure	1-175
#pragma regs_clobbered string	1-176
#pragma regs_clobbered_call string	1-180

#pragma overlay	1-183
#pragma result_alignment (n)	1-184
Class Conversion Optimization Pragas	1-184
#pragma param_never_null param_name [...]	1-184
#pragma suppress_null_check	1-186
Template Instantiation Pragas	1-187
#pragma instantiate instance	1-188
#pragma do_not_instantiate instance	1-189
#pragma can_instantiate instance	1-189
Header File Control Pragas	1-189
#pragma hdrstop	1-189
#pragma no_implicit_inclusion	1-190
#pragma no_pch	1-191
#pragma once	1-192
#pragma system_header	1-192
Inline Control Pragas	1-192
#pragma always_inline	1-192
#pragma inline	1-194
#pragma never_inline	1-194
Linking Control Pragas	1-195
#pragma linkage_name identifier	1-195
#pragma core	1-195
#pragma retain_name	1-200
#pragma section/#pragma default_section	1-201

CONTENTS

#pragma file_attr(“name[=value]” [, “name[=value]” [...]])	1-206
#pragma weak_entry	1-206
Diagnostic Control Pragmas	1-207
Modifying the Severity of Specific Diagnostics	1-207
Modifying the Behavior of an Entire Class of Diagnostics	1-209
Saving or Restoring the Current Behavior of All Diagnostics ..	1-209
Memory Bank Pragmas	1-210
#pragma code_bank(bankname)	1-211
#pragma data_bank(bankname)	1-212
#pragma stack_bank(bankname)	1-213
#pragma bank_memory_kind(bankname, kind)	1-214
#pragma bank_read_cycles(bankname, cycles)	1-215
#pragma bank_write_cycles(bankname, cycles)	1-215
#pragma bank_optimal_width(bankname, width)	1-216
Code Generation Pragmas	1-217
#pragma avoid_anomaly_45 {on off}	1-217
#pragma no_db_return	1-217
Exceptions Table Pragma	1-218
#pragma generate_exceptions_tables	1-218
GCC Compatibility Extensions	1-219
Statement Expressions	1-220
Type Reference Support Keyword (Typeof)	1-221
GCC Generalized Lvalues	1-222
Conditional Expressions with Missing Operands	1-223

Hexadecimal Floating-Point Numbers	1-223
Zero-Length Arrays	1-224
Variable Argument Macros	1-224
Line Breaks in String Literals	1-224
Arithmetic on Pointers to Void and Pointers to Functions	1-225
Cast to Union	1-225
Ranges in Case Labels	1-225
Declarations Mixed With Code	1-225
Escape Character Constant	1-226
Alignment Inquiry Keyword (<code>__alignof__</code>)	1-226
Keyword for Specifying Names in Generated Assembler (<code>asm</code>)	1-226
Function, Variable and Type Attribute Keyword (<code>__attribute__</code>)	1-227
Unnamed struct/union Fields Within struct/unions	1-227
C++ Fractional Type Support	1-228
Format of Fractional Literals	1-228
Conversions Involving Fractional Values	1-229
Fractional Arithmetic Operations	1-229
Mixed Mode Operations	1-230
Saturated Arithmetic	1-230
SIMD Support	1-231
Using SIMD Mode with Multichannel Data	1-233
Using SIMD Mode with Single-Channel Data	1-234
Restrictions to Using SIMD	1-235
<code>SIMD_for</code> Pragma Syntax	1-237

CONTENTS

Compiler Constraints on Using SIMD C/C++	1-238
Impact of Anomaly #40 on SIMD	1-239
Performance When Using SIMD C/C++	1-240
Examples Using SIMD C	1-242
Using SIMD C: Problem Cases—Data Increments	1-242
Using SIMD C: Problem Cases—Data Alignment	1-244
Accessing External Memory on ADSP-2126X and 2136X Processors 1-246	
Link-time Checking of Data Placement	1-246
Inline Functions for External Memory Access	1-247
Support for Interrupts	1-247
Interrupt Dispatchers	1-247
Interrupts and Circular Buffering	1-251
Avoiding Self-Modifying Code	1-251
Interrupt Nesting Restrictions on ADSP-2116x/2126x/ 2136x/2137x Processors	1-252
Restriction on Use of Super-Fast Dispatcher on ADSP-2106x Processors	1-252
Restrictions on using Normal and Circular Buffer Interrupt Dispatchers on 2136x Processors	1-253
Migrating .ldf Files From Previous VisualDSP++ Installations	1-254
C++ Support Tables (ctor, gdt)	1-254
ADSP-21375 Memory Map	1-257
C++ Run-time Libraries Rationalization	1-257
Preprocessor Features	1-258

Predefined Preprocessor Macros	1-259
Writing Macros	1-264
Compound Macros	1-265
C/C++ Run-Time Model and Environment	1-267
C/C++ Run-Time Environment	1-267
Memory Usage	1-270
Program Memory Code Storage	1-271
Data Memory Data Storage	1-272
Program Memory Data Storage	1-272
Run-Time Stack Storage	1-273
Run-Time Heap Storage	1-273
Initialization Data Storage	1-274
Run-Time Header Storage	1-275
Memory Allocation for Stack and Heap on ADSP-2106x, 2116x and 2126x Processors	1-277
Example of Heap/Stack Memory Allocation	1-278
Measuring the Performance of the Compiler	1-279
Constructors and Destructors of Global Class Instances	1-280
Constructors, Destructors and Memory Placement	1-281
Support for argv/argc	1-282
Using Multiple Heaps	1-283
Declaring a Heap	1-284
Heap Identifiers	1-286
Allocating C++ STL Objects to a Non-Default Heap	1-286
Using Alternate Heaps with the Standard Interface	1-289

CONTENTS

Using the Alternate Heap Interface	1-290
C++ Run-Time Support for the Alternate Heap Interface	1-291
Example C Programs	1-291
Compiler Registers	1-293
Miscellaneous Information About Registers	1-294
User Registers	1-294
Call Preserved Registers	1-295
Scratch Registers	1-296
Stack Registers	1-297
Alternate Registers	1-297
Managing the Stack	1-298
Transferring Function Arguments and Return Value	1-304
Passing a C++ Class Instance	1-306
Using Data Storage Formats	1-307
Using the Run-Time Header	1-310
C/C++ and Assembly Interface	1-311
Calling Assembly Subroutines from C/C++ Programs	1-312
Calling C/C++ Functions from Assembly Programs	1-314
Using Mixed C/C++ and Assembly Support Macros	1-316
entry	1-317
exit	1-317
leaf_entry	1-317
leaf_exit	1-317
ccall(x)	1-318

reads(x)	1-318
puts=x	1-318
gets(x)	1-318
alter(x)	1-318
save_reg	1-318
restore_reg	1-319
Using Mixed C/C++ and Assembly Naming Conventions .	1-321
Implementing C++ Member Functions in Assembly Language	1-322
Writing C/C++ Callable SIMD Subroutines	1-324
C++ Programming Examples	1-325
Using Fract Support	1-326
Using Complex Support	1-327
Mixed C/C++/Assembly Programming Examples	1-328
Using Inline Assembly (Add)	1-330
Using Macros to Manage the Stack	1-330
Using Scratch Registers (Dot Product)	1-332
Using Void Functions (Delay)	1-333
Using the Stack for Arguments (Add 5)	1-334
Using Registers for Arguments and Return (Add 2)	1-335
Using Non-Leaf Routines That Make Calls (RMS)	1-336
Using Call Preserved Registers (Pass Array)	1-338
Exceptions Tables in Assembly Routines	1-340
Compiler C++ Template Support	1-344
Template Instantiation	1-344

CONTENTS

Identifying Un-instantiated Templates	1-346
File Attributes	1-348
Automatically-Applied Attributes	1-348
Content Attributes	1-349
FuncName Attributes	1-350
Encoding Attributes	1-350
Default LDF Placement	1-351
Sections versus Attributes	1-352
Granularity	1-352
“Hard” Versus “Soft”	1-352
Number of Values	1-353
Using Attributes	1-353
Example 1	1-353
Example 2	1-355

ACHIEVING OPTIMAL PERFORMANCE FROM C/C++ SOURCE CODE

General Guidelines	2-3
How the Compiler Can Help	2-4
Using the Compiler Optimizer	2-4
Using Compiler Diagnostics	2-5
Warnings and Remarks	2-5
Source and Assembly Annotations	2-7
Using the Statistical Profiler	2-7
Using Profile-Guided Optimization	2-8

Using Profile-Guided Optimization With a Simulator	2-9
Using Profile-Guided Optimization With Non-Simulatable Applications	2-10
Profile-Guided Optimization and Multiple Source Uses .	2-11
Profile-Guided Optimization and the -Ov Switch	2-12
Profile-Guided Optimization and Multiple PGO Data sets	2-12
When to Use Profile-Guided Optimization	2-13
Using Interprocedural Optimization	2-13
Data Types	2-15
Avoiding Emulated Arithmetic	2-16
Getting the Most From IPA	2-17
Initialize Constants Statically	2-17
Dual Word-Aligning Your Data	2-18
Using __builtin_aligned	2-19
Avoiding Aliases	2-21
Indexed Arrays Versus Pointers	2-22
Trying Pointer and Indexed Styles	2-23
Using Function Inlining	2-24
Using Inline asm Statements	2-25
Memory Usage	2-26
Improving Conditional Code	2-28
Loop Guidelines	2-29
Keeping Loops Short	2-30
Avoiding Unrolling Loops	2-30
Avoiding Loop-Carried Dependencies	2-31

CONTENTS

Avoiding Loop Rotation by Hand	2-32
Avoiding Array Writes in Loops	2-33
Inner Loops vs. Outer Loops	2-33
Avoiding Conditional Code in Loops	2-34
Avoiding Placing Function Calls in Loops	2-35
Avoiding Non-Unit Strides	2-35
Loop Control	2-36
Using the Restrict Qualifier	2-37
Avoiding Long Latencies	2-38
Using Built-In Functions in Code Optimization	2-39
Using System Support Built-In Functions	2-39
Using Circular Buffers	2-40
Smaller Applications: Optimizing for Code Size	2-43
Using Pragmas for Optimization	2-45
Function Pragmas	2-45
#pragma alloc	2-45
#pragma const	2-46
#pragma pure	2-46
#pragma result_alignment	2-47
#pragma regs_clobbered	2-47
#pragma optimize_{off for_speed for_space as_cmd_line} ..	2-49
Loop Optimization Pragmas	2-50
#pragma loop_count	2-50
#pragma no_vectorization	2-51

#pragma vector_for	2-51
#pragma SIMD_for	2-52
#pragma all_aligned	2-52
#pragma no_alias	2-53
Useful Optimization Switches	2-54
How Loop Optimization Works	2-55
Terminology	2-55
Clobbered Register	2-55
Live Register	2-56
Spill	2-56
Scheduling	2-56
Loop Kernel	2-56
Loop Prolog	2-57
Loop Epilog	2-57
Loop Invariant	2-57
Hoisting	2-57
Sinking	2-58
Loop Optimization Concepts	2-58
Software Pipelining	2-59
Loop Rotation	2-59
Loop Vectorization	2-62
Modulo Scheduling	2-64
Initiation Interval (II) and the kernel	2-65
Minimum Initiation Interval Due to Resources (Res MII)	2-68

Minimum Initiation Interval Due to Recurrences (Rec MII)	2-68
Stage Count (SC)	2-69
Variable Expansion and MVE Unroll	2-71
Trip Count	2-76
A Worked Example	2-77
Assembly Optimizer Annotations	2-80
Global Information	2-81
Procedure Statistics	2-82
Instruction Annotations	2-87
Loop Identification	2-87
Loop Identification Annotations	2-88
File Position	2-91
Vectorization	2-94
Loop Flattening	2-95
Vectorization Annotations	2-96
Modulo Scheduling Information	2-98
Annotations for Modulo Scheduled Instructions	2-99
Warnings, Failure Messages and Advice	2-105

INDEX

PREFACE

Thank you for purchasing Analog Devices, Inc. development software for signal processing applications.

Purpose of This Manual

The *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* contains information about the C/C++ compiler and its features designed for use with SHARC[®] (ADSP-21xxx) processors. It includes syntax for command lines, switches, and language extensions. It leads you through the process of using library routines and writing mixed C/C++/assembly code.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. This manual assumes that the audience has a working knowledge of the SHARC architecture and instruction set and the C/C++ programming languages.

Programmers who are unfamiliar with SHARC processors can use this manual, but they should supplement it with other texts (such as the appropriate hardware reference and programming reference manuals) that describe their target architectures.

Manual Contents

This manual contains:

- Chapter 1, “[Compiler](#)”
Provides information on compiler options, language extensions and C/C++/assembly interfacing
- Chapter 2, “[Achieving Optimal Performance from C/C++ Source Code](#)”
Shows how to optimize compiler operation

What’s New in This Manual

This is an updated manual. The *VisualDSP++ 5.0 C/C++ Compiler Manual for SHARC Processors* provides information on C/C++ compiler and its features and documents support for all current SHARC processors. It does not describe C/C++ and DSP run-time libraries which are separated into a library reference manual, *VisualDSP++ 5.0 C/C++ Run-Time Library Manual for SHARC Processors*.

Refer to the *VisualDSP++ 5.0 Product Release Bulletin* for a complete list of new compiler features and enhancements.

Technical or Customer Support

You can reach Analog Devices, Inc. Customer Support in the following ways:

- Visit the Embedded Processing and DSP products Web site at http://www.analog.com/processors/technical_support
- E-mail tools questions to processor.tools.support@analog.com

- E-mail processor questions to
processor.support@analog.com (World wide support)
processor.europe@analog.com (Europe support)
processor.china@analog.com (China support)
- Phone questions to **1-800-ANALOGD**
- Contact your Analog Devices, Inc. local sales office or authorized distributor
- Send questions by mail to:
Analog Devices, Inc.
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
USA

Supported Processors

The name “SHARC” refers to a family of Analog Devices, Inc. high-performance 32-bit floating-point digital signal processors that can be used in speech, sound, graphics, and imaging applications. For a complete list of processors supported by VisualDSP++ 5.0, refer to VisualDSP++ online Help.

Product Information

Product information can be obtained from the Analog Devices Web site, VisualDSP++ online Help system, and a technical library CD.

Analog Devices Web Site

The Analog Devices Web site, www.analog.com, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

To access a complete technical library for each processor family, go to http://www.analog.com/processors/technical_library. The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Visit MyAnalog.com to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

VisualDSP++ Online Documentation

Online documentation comprises the VisualDSP++ Help system, software tools manuals, hardware tools manuals, processor manuals, Dinkum Abridged C++ library, and FLEXnet License Tools documentation. You can search easily across the entire VisualDSP++ documentation set for any topic of interest.

For easy printing, supplementary Portable Documentation Format (.pdf) files for all manuals are provided on the VisualDSP++ installation CD.

Each documentation file type is described as follows.

File	Description
.chm	Help system files and manuals in Microsoft help format
.htm or .html	Dinkum Abridged C++ library and FLEXnet license tools software documentation. Viewing and printing the .html files requires a browser, such as Internet Explorer 6.0 (or higher).
.pdf	VisualDSP++ and processor manuals in PDF format. Viewing and printing the .pdf files requires a PDF reader, such as Adobe Acrobat Reader (4.0 or higher).

Technical Library CD

The technical library CD contains seminar materials, product highlights, a selection guide, and documentation files of processor manuals, VisualDSP++ software manuals, and hardware tools manuals for the following processor families: Blackfin, SHARC, TigerSHARC, ADSP-218x, and ADSP-219x.

To order the technical library CD, go to http://www.analog.com/processors/technical_library, navigate to the manuals page for your processor, click the request CD check mark, and fill out the order form.

Data sheets, which can be downloaded from the Analog Devices Web site, change rapidly, and therefore are not included on the technical library CD. Technical manuals change periodically. Check the Web site for the latest manual revisions and associated documentation errata.




Notation Conventions

Text conventions used in this manual are identified and described as follows.



Additional conventions, which apply only to specific chapters, may appear throughout this document.

Notation Conventions

Example	Description
Close command (File menu)	Titles in in bold style reference sections indicate the location of an item within the VisualDSP++ environment's menu system (for example, the Close command appears on the File menu).
{this that}	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as <i>this</i> or <i>that</i> . One or the other is required.
[this that]	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional <i>this</i> or <i>that</i> .
[this,...]	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of <i>this</i> .
.SECTION	Commands, directives, keywords, and feature names are in text with letter gothic font.
<i>filename</i>	Non-keyword placeholders appear in text with italic style format.
	Note: For correct operation, ... A Note provides supplementary information on a related topic. In the online version of this book, the word Note appears instead of this symbol.
	Caution: Incorrect device operation may result if ... Caution: Device damage may result if ... A Caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word Caution appears instead of this symbol.
	Warning: Injury to device users may result if ... A Warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word Warning appears instead of this symbol.

1 COMPILER

The C/C++ compiler (`cc21k`) is part of Analog Devices development software for SHARC (ADSP-21xxx) processors.



The code examples in this manual have been compiled using VisualDSP++ 5.0. The examples compiled with other versions of VisualDSP++ may result in build errors or different output although the highlighted algorithms stand and should continue to stand in future releases of VisualDSP++.

This chapter contains:

- [“C/C++ Compiler Overview” on page 1-3](#)
provides an overview of C/C++ compiler for SHARC processors.
- [“Compiler Command-Line Interface” on page 1-7](#)
describes the operation of the compiler as it processes programs, including input and output files, and command-line switches.
- [“MISRA-C Compiler” on page 1-91](#)
describes how the `cc21k` compiler enables checking for MISRA-C: 2004 Guidelines.
- [“C/C++ Compiler Language Extensions” on page 1-104](#)
describes the `cc21k` compiler’s extensions to the ISO/ANSI standard for the C and C++ languages.
- [“Preprocessor Features” on page 1-258](#)
contains information on the preprocessor and ways to modify source compilation.

- [“C/C++ Run-Time Model and Environment” on page 1-267](#)
contains reference information about implementation of C/C++ programs, data, and function calls in ADSP-21xxx processors.
- [“C/C++ and Assembly Interface” on page 1-311](#)
describes how to call an assembly language subroutine from a C/C++ program, and how to call a C/C++ function from within an assembly language program.
- [“Compiler C++ Template Support” on page 1-344](#)
describes how templates are instantiated at compile time
- [“File Attributes” on page 1-348](#)
describes how file attributes help with the placement of runtime library functions.

C/C++ Compiler Overview

The C/C++ compiler (`cc21k`) is designed to aid your project development efforts by:

- Processing C and C++ source files, producing machine-level versions of the source code and object files
- Providing relocatable code and debugging information within the object files
- Providing relocatable data and program memory segments for placement by the linker in the processors' memory

Using C/C++, developers can significantly decrease time-to-market since it gives them the ability to efficiently work with complex signal processing data types. It also allows them to take advantage of specialized processor operations without having to understand the underlying processor architecture.

The C/C++ compiler (`cc21k`) compiles ISO/ANSI standard C and C++ code for the SHARC processors. Additionally, Analog Devices includes within the compiler a number of C language extensions designed to assist in processor development. The compiler runs from the VisualDSP++ environment or from an operating system command line.

The C/C++ compiler (`cc21k`) processes your C and C++ language source files and produces SHARC assembler source files. The assembler source files are assembled by the SHARC assembler (`eam21k`). The assembler creates Executable and Linkable Format (ELF) object files that can either be linked (using the linker) to create an ADSP-21xxx executable file or included in an archive library (`elfar`). The way in which the compiler controls the assemble, link, and archive phases of the process depends on the source input files and the compiler options used.

C/C++ Compiler Overview

Your source files contain the C/C++ program to be processed by the compiler. The cc21k compiler supports the ANSI/ISO standard definitions of the C and C++ languages. RTTI and Exceptions for C++ are supported, but disabled by default. See information on these switches: [“-rtti” on page 1-76](#) and [“-eh” on page 1-32](#).

For information on the C language standard, see any of the many reference texts on the C language. Analog Devices recommends the Bjarne Stroustrup text *“The C++ Programming Language”* from Addison Wesley Longman Publishing Co (ISBN: 0201889544) (1997) as a reference text for the C++ programming language.

The cc21k compiler supports a set of C/C++ language extensions. These extensions support hardware features of the ADSP-21xxx processors. For information on these extensions, see [“C/C++ Compiler Language Extensions” on page 1-104](#).

You can set the compiler options from the **Compile** page of the **Project Options** dialog box of the VisualDSP++ Integrated Development and Debug Environment (IDDE). These selections control how the compiler processes your source files, letting you select features that include the language dialect, error reporting, and debugger output.

For more information on the VisualDSP++ environment, see the *VisualDSP++ 5.0 User's Guide* and online Help.

Standard Conformance

Analog C compilers conform to the ISO/IEC 998:1990 C standard and the ISO/IEC 14882:1998 C++ standard (in C++ mode) with a small number of currently unsupported features or areas of divergence.

Unsupported features are:

- Runtime-type information (RTTI) for C++
- Exceptions for C++
- ANSI features that require operating-system support are generally not supported. This includes `time.h` functionality in C.

Areas of divergence from standard conformance are:

- The `double` type is defined in terms of a single precision 32-bit floats, not double precision 64-bit floats.
- Normal ANSI C external linkage does not specifically require standard include files to be used, although it is recommended. In most cases, Analog C compilers do require standard include files. This is because build configurations and optimization levels are used to select the correct and optimal implementation of the functions. For example, the include files may redefine standard C functions to use optimal compiler built-in implementations.

The compilers also support a number of language extensions that are essentially aids to DSP programmers and would not be defined in strict ANSI conforming implementations. These extensions are usually enabled by default and in some cases can be disabled using a command-line switch, if required. These extensions include:

- Inline (function), which directs the compiler to integrate the function code into the code of the callers.
- Dual memory support keywords (pm/dm). Disabled using the [“-no-extra-keywords” on page 1-45](#).
- Placement support keyword (section). Disabled using the [“-no-extra-keywords” on page 1-45](#)..
- Boolean type support keywords in C (bool, true, false). Disabled using the [“-no-extra-keywords” on page 1-45](#).
- Variable length array support.
- Non-constant aggregate initializer support.
- Indexed initializer support.
- Preprocessor generated warnings.
- Support for C++-style comments in C programs.
- For more information on these extensions, see [“C/C++ Compiler Language Extensions” on page 1-104](#).

Compiler Command-Line Interface

This section describes how the `cc21k` compiler is invoked from the command line, the various types of files used by and generated from the compiler, and the switches used to tailor the compiler's operation.

This section contains:

- [“Running the Compiler” on page 1-8](#)
- [“Compiler Command-Line Switches” on page 1-9](#)
- [“Environment Variables Used by the Compiler” on page 1-77](#)
- [“Data Type and Data Type Sizes” on page 1-78](#)
- [“Optimization Control” on page 1-80](#)
- [“Controlling Silicon Revision and Anomaly Workarounds within the Compiler” on page 1-85](#)

By default, the compiler runs with Analog Devices extension keywords for C code enabled. This means that the compiler processes source files written in ANSI/ISO standard C language supplemented with Analog Devices extensions. [Table 1-2 on page 1-9](#) lists valid extensions of source files the compiler operates upon. By default, the compiler processes the input file through the listed stages to produce a `.DXE` file. (See file names in [Table 1-3 on page 1-10](#).) [Table 1-2 on page 1-9](#) lists the switches that select the language dialect.

Although many switches are generic between C and C++, some of them are valid in C++ mode only. A summary of the generic C/C++ compiler switches appears in [Table 1-3 on page 1-10](#). A summary of the C++-specific compiler switches appears in [Table 1-5 on page 1-21](#). The summaries are followed by descriptions of each switch.



When developing a project, sometimes it is useful to modify the compiler's default options settings. The way the compiler's options are set depends on the environment used to run the processor development software.

Running the Compiler

Use the following syntax for the `cc21k` command line:

```
cc21k [-switch [-switch ...] sourcefile [sourcefile ...]]
```

[Table 1-1](#) describes these syntax elements.

Table 1-1. `cc21k` Command Line Syntax

Command Element	Description
<code>cc21k</code>	Name of the compiler program for SHARC processors.
<code>-switch</code>	Switch (or switches) to process. The compiler has many switches. These switches select the operations and modes for the compiler and other tools. Command-line switches are case sensitive. For example, <code>-0</code> is not the same as <code>-o</code> .
<code>sourceFile</code>	Name of the file to be preprocessed, compiled, assembled, and/or linked



When file names or other switches for the compiler include spaces or other special characters, you must ensure that these are properly quoted (usually using double-quote characters), to ensure that they are not interpreted by the operating system before being passed to the compiler.

The `sourceFile` element (the name of the source file to be processed) can include the drive, directory, file name and file extension. The compiler supports both Win32 and POSIX-style paths by using forward or back slashes as the directory delimiter. It also supports UNC path names (starting with two slashes and a network name).

If the name contains spaces, enclose it in straight quotes; for example, "long file name.c". The cc21k compiler uses the file extension to determine what the file contains ([Table 1-3 on page 1-10](#)) and what operations to perform upon it ([Table 1-2 on page 1-9](#)).

Compiler Command-Line Switches

This section describes the command-line switches used when compiling. It contains a set of tables that provide a brief description of each switch. These tables are organized by type of switch. Following these tables are sections that provide fuller descriptions of each switch.

C/C++ Compiler Switch Summaries

This section contains a set of tables that summarize generic and specific switches (options).

- “C/C++ Mode Selection Switches”, [Table 1-2](#)
- “C/C++ Compiler Common Switches”, [Table 1-3](#)
- “C Mode (MISRA) Compiler Switches”, [Table 1-4 on page 1-20](#)
- “C++ Mode Compiler Switches”, [Table 1-5 on page 1-21](#)

A brief description of each switch follows the tables, beginning [on page 1-22](#).

Table 1-2. C/C++ Mode Selection Switches

Switch Name	Description
-c89 (on page 1-22)	Supports programs that conform to the ISO/IEC 9899:1990 standard
-c++ (on page 1-22)	Supports ANSI/ISO standard C++ with Analog Devices extensions. Note that C++ is not supported on the ADSP-21020 processor.

Compiler Command-Line Interface

Table 1-3. C/C++ Compiler Common Switches

Switch Name	Description
<code>sourcefile</code> (on page 1-23)	Specifies file to be compiled
<code>-@ filename</code> (on page 1-23)	Reads command-line input from the file
<code>-A name[tokens]</code> (on page 1-23)	Asserts the specified name as a predicate
<code>-add-debug-libpaths</code> (on page 1-24)	Link against debug-specific variants of system libraries, where available.
<code>-aligned-stack</code> (on page 1-25)	Aligns the program stack on a double-word boundary
<code>-alttok</code> (on page 1-25)	Allows alternative keywords and sequences in sources
<code>-always-inline</code> (on page 1-26)	Treats <code>inline</code> keyword as a requirement rather than a suggestion.
<code>-annotate</code> (on page 1-26)	Annoates compiler-produced assembly files
<code>-annotate-loop-instr</code> (on page 1-26)	Provides additional annotation information for the prolog, kernel and epilog of a loop
<code>-auto-attrs</code> (on page 1-34)	Directs the compiler to emit automatic attributes based on the files it compiles. Enabled by default.
<code>-build-lib</code> (on page 1-27)	Directs the librarian to build a library file
<code>-C</code> (on page 1-27)	Retains preprocessor comments in the output file; must run with the <code>-E</code> or <code>-P</code> switch
<code>-c</code> (on page 1-27)	Compiles and/or assembles only, but does not link
<code>-compatible-pm-dm</code> (on page 1-27)	Specifies that the compiler shall treat <code>dm-</code> and <code>pm-</code> qualified pointers as assignment-compatible
<code>-const-read-write</code> (on page 1-28)	Specifies that data accessed via a pointer to <code>const</code> data may be modified elsewhere
<code>-const-strings</code> (on page 1-28)	Directs the compiler to mark string literals as const-qualified

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
<code>-D macro[=definition]</code> (on page 1-28)	Defines a macro
<code>-debug-types</code> (on page 1-29)	Supports building a *.h file directly and writing a complete set of debugging information for the header file
<code>-double-size [-32 -64]</code> (on page 1-29)	Selects 32- or 64-bit IEEE format for double. The <code>-double-size-32</code> is the default mode.
<code>-double-size-any</code> (on page 1-30)	Indicates that the resulting object can be linked with objects built with any double size
<code>-dry</code> (on page 1-31)	Displays, but does not perform, main driver actions (verbose dry-run)
<code>-dryrun</code> (on page 1-31)	Displays, but does not perform, top-level driver actions (terse dry-run)
<code>-E</code> (on page 1-31)	Preprocesses, but does not compile, the source file
<code>-ED</code> (on page 1-31)	Preprocesses and sends all output to a file
<code>-EE</code> (on page 1-31)	Preprocesses and compiles the source file
<code>-eh</code> (on page 1-32)	Enables exception handling
<code>-enum-is-int</code> (on page 1-33)	By default enums can have a type larger than <code>int</code> . This option ensures the <code>enum</code> type is <code>int</code> .
<code>-extra-keywords</code> (on page 1-33)	Recognizes ADI extensions to ANSI/ISO standards for C and C++ (default mode)
<code>-file-attr name[=value]</code> (on page 1-34)	Adds the specified attribute name/value pair to the file(s) being compiled
<code>-flags-{tools} <arg1> [,arg2...]</code> (on page 1-34)	Passes command-line switches through the compiler to other build tools
<code>-float-to-int</code> (on page 1-34)	Uses a support library function to convert a <code>float</code> to an integer

Compiler Command-Line Interface

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
<code>-force-circbuf</code> (on page 1-34)	Treats array references of the form <code>array[i%n]</code> as circular buffer operations
<code>-fp-associative</code> (on page 1-35)	Treats floating-point multiply and addition as an associative
<code>-full-version</code> (on page 1-35)	Displays the version number of the driver and any processes invoked by the driver
<code>-g</code> (on page 1-35)	Generates DWARF-2 debug information
<code>-glite</code> (on page 1-36)	Generates lightweight DWARF-2 debug information
<code>-H</code> (on page 1-36)	Outputs a list of included header files, but does not compile
<code>-HH</code> (on page 1-36)	Outputs a list of included header files and compiles
<code>-h[elp]</code> (on page 1-36)	Outputs a list of command-line switches
<code>-I <i>directory</i></code> (on page 1-36)	Appends directory to the standard search path
<code>-I-</code> (on page 1-37)	Establishes the point in the <code>include</code> directory list at which the search for header files enclosed in angle brackets should begin
<code>-i</code> (on page 1-38)	Outputs only header details or makefile dependencies for <code>include</code> files specified in double quotes
<code>-implicit-pointers</code> (on page 1-38)	Demotes incompatible-pointer-type errors into discretionary warnings. Not valid when compiling in C++ mode.
<code>-include <i>filename</i></code> (on page 1-39)	Includes named file prior to preprocessing each source file
<code>-ipa</code> (on page 1-39)	Enables interprocedural analysis
<code>-L <i>directory</i></code> (on page 1-39)	Appends directory to the standard library search path

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
<code>-l <i>library</i></code> (on page 1-39)	Searches library for functions when linking
<code>-list-workarounds</code> (on page 1-40)	Lists all compiler-supported errata workarounds
<code>-M</code> (on page 1-40)	Generates make rules only, but does not compile
<code>-MD</code> (on page 1-40)	Generates make rules, compiles, and prints to a file
<code>-MM</code> (on page 1-41)	Generates make rules and compiles
<code>-Mo <i>filename</i></code> (on page 1-41)	Writes dependency information to <i>filename</i> . This switch is used in conjunction with the <code>-ED</code> or <code>-MD</code> options
<code>-Mt <i>filename</i></code> (on page 1-41)	Makes dependencies, where the target is renamed as <i>filename</i>
<code>-map <i>filename</i></code> (on page 1-41)	Directs the linker to generate a memory map of all symbols
<code>-mem</code> (on page 1-41)	Enables memory initialization
<code>-misra</code> (on page 1-71)	(C compiler switch): Enables checking for MISRA-C: 2004 Guidelines, allows some relaxation of interpretation.
<code>-multiline</code> (on page 1-41)	Enables string literals over multiple lines (default)
<code>-never-inline</code> (on page 1-42)	Ignores <code>inline</code> keyword on function definitions
<code>-no-aligned-stack</code> (on page 1-42)	Does not double-word align the program stack
<code>-no-alttok</code> (on page 1-42)	Does not allow alternative keywords and sequences in sources
<code>-no-annotate</code> (on page 1-42)	Disables the annotation of assembly files

Compiler Command-Line Interface

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
-no-annotate-loop-instr (on page 1-43)	Disables the production of additional loop annotation information by the compiler (default mode)
-no-auto-attrs (on page 1-34)	Directs the compiler not to emit automatic attributes based on the files it compiles.
-no-builtin (on page 1-43)	Recognizes only built-in functions that begin with two underscores(__)
-no-circbuf (on page 1-44)	Disables the automatic generation of circular buffer code by the compiler
-no-db (on page 1-44)	Specifies that the compiler shall not generate code containing delayed branches jumps
-no-defs (on page 1-44)	Disables preprocessor definitions: macros, include directories, library directories, run-time headers, or keyword extensions
-no-eh (on page 1-45)	Disables exception handling
-no-extra-keywords (on page 1-45)	Does not accept ADI keyword extensions that might affect ISO/ANSI standards for C and C++
-no-fp-associative (on page 1-45)	Does not treat floating-point multiply and addition as an associative
-no-mem (on page 1-45)	Disables memory initialization
-no-multiline (on page 1-46)	Disables multiple line string literal support
-no-progress-rep-timeout (on page 1-46)	Prevents the compiler from issuing a diagnostic during excessively long compilations.
-no-sat-associative (on page 1-46)	Saturating addition is not associative.
-no-saturation (on page 1-46)	Causes the compiler not to introduce saturation semantics when optimizing expressions
-no-simd (on page 1-47)	Disables automatic SIMD mode when compiling for ADSP-2116x, ADSP-2126x or ADSP-213xx processors

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
<code>-no-std-ass</code> (on page 1-47)	Disables any predefined assertions and system-specific macro definitions
<code>-no-std-def</code> (on page 1-48)	Disables preprocessor definitions and ADI keyword extensions that do not have leading underscores(__)
<code>-no-std-inc</code> (on page 1-48)	Searches for preprocessor include header files only in the current directory and in directories specified with the <code>-I</code> switch
<code>-no-std-lib</code> (on page 1-48)	Searches for only those library files specified with the <code>-l</code> switch
<code>-no-threads</code> (on page 1-48)	Specifies that all compiled code need not be thread-safe
<code>-no-workaround workaround_id</code> (on page 1-48)	Disables specific hardware anomaly workarounds within the compiler.
<code>-normal-word-code</code> (on page 1-49)	Directs the compiler to generate instructions of normal-word size (48-bits)
<code>-nwc</code> (on page 1-49)	Has the same effect as compiling with the <code>-normal-word-code</code> switch
<code>-O [0 1]</code> (on page 1-49)	Enables code optimizations
<code>-Oa</code> (on page 1-49)	Enables automatic function inlining
<code>-Og</code> (on page 1-50)	Enables a compiler mode that performs optimizations while still preserving the debugging information
<code>-Os</code> (on page 1-50)	Optimizes for code size
<code>-Ov num</code> (on page 1-50)	Controls speed versus size optimizations
<code>-o filename</code> (on page 1-52)	Specifies the output file name
<code>-overlay</code> (on page 1-52)	Disables the propagation of register information between functions and forces the compiler to assume that all functions clobber all scratch registers

Compiler Command-Line Interface

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
-overlay-clobbers <i>regs</i> (on page 1-53)	Specifies the registers assumed to be clobbered by an overlay manager.
-P (on page 1-53)	Preprocesses, but does not compile, the source file; omits line numbers in the preprocessor output
-PP (on page 1-54)	Similar to -P, but does not halt compilation after pre-processing
-path-{asm compiler lib link} <i>pathname</i> (on page 1-54)	Uses the specified directory as the location of the specified compilation tool (assembler, compiler, librarian, or linker, respectively)
-path-install <i>directory</i> (on page 1-54)	Uses the specified directory as the location of all compilation tools
-path-output <i>directory</i> (on page 1-54)	Specifies the location of non-temporary files
-path-temp <i>directory</i> (on page 1-54)	Specifies the location of temporary files
-pch (on page 1-55)	Generates and uses precompiled header files (*.pch)
-pchdir <i>directory</i> (on page 1-55)	Specifies the location of PCHRepository
-pguide (on page 1-56)	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization
-pplist <i>filename</i> (on page 1-56)	Outputs a raw preprocessed listing to the specified file
-proc <i>processor</i> (on page 1-57)	Specifies that the compiler should produce code suitable for the specified processor
-progress-rep-func (on page 1-57)	Issues a diagnostic message each time the compiler starts compiling a new function. Equivalent to -Wwarn=cc1472.
-progress-rep-opt (on page 1-58)	Issues a diagnostic message each time the compiler starts a new generic optimization pass on the current function. Equivalent to -Wwarn=cc1473.

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
-progress-rep-timeout (on page 1-58)	Issues a diagnostic message if the compiler exceeds a time limit during compilation
-progress-rep-timeout-secs <i>secs</i> (on page 1-58)	Specifies how many seconds must elapse during a compilation before the compiler issues a diagnostic on the length of compilation
-R <i>directory</i> (on page 1-58)	Appends directory to the standard search path for source files
-R- (on page 1-59)	Removes all directories from the standard search path for source files
-reserve <reg1>[,reg2...] (on page 1-59)	Reserves certain registers from compiler use. Note: Reserving registers can have a detrimental effect on the compiler's optimization capabilities.
-restrict-hardware-loops <i>maximum</i> (on page 1-59)	Restrict the number of levels of loop nesting used by the compiler
-S (on page 1-60)	Stops compilation before running the assembler
-s (on page 1-60)	Removes debug info from the output executable file
-sat-associative (on page 1-60)	Saturating addition is associative
-save-temps (on page 1-60)	Saves intermediate files
-section id=section_name (on page 1-60)	Orders the compiler to place data/program of type "id" into the section "section_name"
-short-word-code (on page 1-62)	Directs the compiler to generate instructions of short word size (16/32/48-bits)
-show (on page 1-62)	Displays the driver command-line information
-si-revision <i>version</i> (on page 1-62)	Specifies a silicon revision of the specified processor. The default setting is the latest silicon revision.
-signed-bitfield (on page 1-62)	Makes the default type for <code>int</code> bit-fields signed

Compiler Command-Line Interface

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
-structs-do-not-overlap (on page 1-63)	Specifies that <code>struct</code> copies may use “memcpy” semantics, rather than the usual “memmove” behavior
-swc (on page 1-63)	Directs the compiler to generate instructions of short word size (16/32/48-bits)
-syntax-only (on page 1-64)	Checks the source code for compiler syntax errors, but does not write any output
-sysdefs (on page 1-64)	Defines the system definition macros
-T <i>filename</i> (on page 1-64)	Specifies the Linker Description File
-threads (on page 1-64)	Specifies that support for multithreaded applications is to be enabled
-time (on page 1-65)	Displays the elapsed time as part of the output information on each part of the compilation process
-U <i>macro</i> (on page 1-65)	Undefines macro(s)
-unsigned-bitfield (on page 1-65)	Makes the default type for plain <code>int</code> bit-fields unsigned
-v (on page 1-66)	Displays both the version and command-line information
-verbose (on page 1-66)	Displays command-line information
-version (on page 1-66)	Displays version information
-W{error remark suppress warn} <i>number</i> (on page 1-67)	Overrides the default severity of the specified error message
-Werror-limit <i>number</i> (on page 1-67)	Stops compiling after reaching the specified number of errors
-Werror-warnings (on page 1-67)	Directs the compiler to treat all warnings as errors

Table 1-3. C/C++ Compiler Common Switches (Cont'd)

Switch Name	Description
-Wremarks (on page 1-68)	Indicates that the compiler may issue remarks, which are diagnostic messages even milder than warnings
-Wterse (on page 1-68)	Issues only the briefest form of compiler warning, errors, and remarks.
-W (on page 1-68)	Does not display compiler warning messages
-warn-protos (on page 1-68)	Produces a warnings when a function is called without a prototype
-workaround <i>workaround_id</i> (on page 1-69)	Enables code generator workaround for specific hardware errata
-write-files (on page 1-69)	Enables compiler I/O redirection
-write-opts (on page 1-69)	Passes the user options (but not input filenames) via a temporary file
-xref <i>filename</i> (on page 1-70)	Outputs cross-reference information to the specified file

Table 1-4. C Mode (MISRA) Compiler Switches

Switch Name	Description
<code>-misra</code> (on page 1-71)	Enables checking for MISRA-C: 2004 Guidelines, allows some relaxation of interpretation
<code>-misra-linkdir</code> (on page 1-71)	Specifies directory for generation of <code>.misra</code> files. If this option is not specified, a local directory called <code>MISRARepository</code> is created.
<code>-misra-no-cross-module</code> (on page 1-71)	Enables checking for MISRA-C: 2004 Guidelines, allows some relaxation of interpretation. Does not generate <code>.misra</code> files to check for link-time rule violations.
<code>-misra-no-runtime</code> (on page 1-71)	Enables checking for MISRA-C: 2004 Guidelines, allows some relaxation of interpretation. Does not generate extra code to perform run-time checking in support of a number of Rules.
<code>-misra-strict</code> (on page 1-72)	Enables checking for MISRA-C: 2004 Guidelines
<code>-misra-suppress-advisory</code> (on page 1-72)	Enables checking for MISRA-C: 2004 Guidelines. Advisory rules are not reported.
<code>-misra-testing</code> (on page 1-72)	Enables checking for MISRA-C: 2004 Guidelines. Suppresses reporting of MISRA-C rule 20.4, 20.7, 20.8, 20.9, 20.10, 20.11 and 20.12.
<code>-Wmis_suppress</code> (on page 1-72)	Overrides the default severity of the specified messages relating to the specified MISRA-C rules
<code>-Wmis_warn</code> (on page 1-73)	Overrides the default severity of the specified messages relating to the specified MISRA-C rules

Table 1-5. C++ Mode Compiler Switches

Switch Name	Description
-anach (on page 1-73)	Supports some language features (anachronisms) that are prohibited by the C++ standard but still in common use
-check-init-order (on page 1-74)	Adds run-time checking to the generated code highlighting potential uninitialized external objects.
-full-dependency-inclusion (on page 1-75)	Ensures re-inclusion of implicitly included files when generating dependency information
-ignore-std (on page 1-75)	Disables namespace <code>std</code> within the C++ Standard header files.
-no-anach (on page 1-76)	Disallows the use of anachronisms that are prohibited by the C++ standard
-no-implicit-inclusion (on page 1-76)	Prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated
-no-rtti (on page 1-76)	Disables run-time type information
-no-std-templates (on page 1-76)	Disables the lookup of names used in templates
-rtti (on page 1-76)	Enables run-time type information
-std-templates (on page 1-77)	Enables the lookup of names used in templates

C/C++ Mode Selection Switch Descriptions

The following command-line switches provide C/C++ mode selection.

-c89

The `-c89` switch directs the compiler to support programs that conform to the ISO/IEC 9899:1990 standard. For greater conformance to the standard, the following switches should be used: `-alttok`, `-const-read-write`, and `-no-extra-keywords`. (See [Table 1-3 on page 1-10.](#))

-C++

The `-c++` (C++ mode) switch directs the compiler to compile the source file(s) written in ANSI/ISO standard C++ with Analog Devices language extensions. When using this switch, source files with an extension of `.c` is compiled and linked in C++ mode.

All the standard features of C++ are accepted in the default mode except exception handling and run-time type identification because these impose a run-time overhead that is not desirable for all embedded programs. Support for these features can be enabled with the `-eh` and `-rtti` switches. (See [Table 1-5.](#))



C++ is not supported on the ADSP-21020 processor.

C/C++ Compiler Common Switch Descriptions

The following command-line switches apply in both C and C++ modes.

sourcefile

The *sourcefile* parameter (or parameters) specifies the name of the file (or files) to be preprocessed, compiled, assembled, and/or linked. A file name can include the drive, directory, file name, and file extension. The cc21k compiler uses the file extension to determine the operations to perform. [Table 1-3 on page 1-10](#) lists the permitted extensions and matching compiler operations.

-@ filename

The *-@filename* (command file) switch directs the compiler to read command-line input from *filename*. The specified file must contain driver options but may also contain source filenames and environment variables. It can be used to store frequently used options as well as to read from a file list

-A name[tokens]

The *-A* (assert) switch directs the compiler to assert *name* as a predicate with the specified *tokens*. This has the same effect as the *#assert* preprocessor directive. The following assertions are predefined:

Table 1-6. Predefined Assertions

Assertion	Value
system	embedded
machine	adsp21xxx
cpu	adsp21xxx
compiler	cc21k

Compiler Command-Line Interface

The `-A name(value)` switch is equivalent to including

```
#assert name(value)
```

in your source file, and both may be tested in a preprocessor condition in the following manner:

```
#if #name(value)
    // do something
#else
    // do something else
#endif
```

For example, the default assertions may be tested as:

```
#if #machine(adsp21xxx)
    // do something
#endif
```



The parentheses in the assertion need quotes when using the `-A` switch, to prevent misinterpretation. No quotes are needed for a `#assert` directive in a source file.

-add-debug-libpaths

The `-add-debug-libpaths` switch prepends the `Debug` subdirectory to the search paths passed to the linker. The `Debug` subdirectory, found in each of the silicon-revision-specific library directories, contains variants of certain libraries (for example, system services), which provide additional diagnostic output to assist in debugging problems arising from their use.



Invoke this switch with the **Use Debug System Libraries** radio button located in the VisualDSP++ **Project Options** dialog box, **Link** page, **Processor** category.

-aligned-stack

The `-aligned-stack` switch directs the compiler to align the program stack on a double-word boundary.

-alttok

The `-alttok` (alternative tokens) switch directs the compiler to allow alternative operator keywords and digraph sequences in source files. Additionally, this switch enables the recognition of these alternative operator keywords in C++ source files:

Table 1-7. Alternative Operator Keywords

Keyword	Equivalent
<code>and</code>	<code>&&</code>
<code>and_eq</code>	<code>&=</code>
<code>bitand</code>	<code>&</code>
<code>bitor</code>	<code> </code>
<code>compl</code>	<code>~</code>
<code>or</code>	<code> </code>
<code>or_eq</code>	<code> =</code>
<code>not</code>	<code>!</code>
<code>not_eq</code>	<code>!=</code>
<code>xor</code>	<code>^</code>
<code>xor_eq</code>	<code>^=</code>

See also the `-no-alttok` switch ([on page 1-42](#)).



To use alternative tokens in C, you should use `#include <iso646.h>`.

-always-inline

The `-always-inline` switch instructs the compiler to always attempt to inline any call to a function that is defined with the `inline` qualifier. It is equivalent to applying `#pragma always_inline` to all functions in the module that have the `inline` qualifier. See also the `-never-inline` switch ([on page 1-42](#)).



Invoke this switch with the **Always** radio button located in the Inlining area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-annotate

The `-annotate` (enable assembly annotations) switch directs the compiler to annotate assembly files generated by the compiler. The default behavior is that whenever optimizations are enabled all assembly files generated by the compiler are annotated with information on the performance of the generated assembly. See “[Assembly Optimizer Annotations](#)” [on page 2-80](#) for more details on this feature. Also, see also the `-no-annotate` switch ([on page 1-42](#)).



Invoke this switch by checking the **Generate assembly code annotations** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-annotate-loop-instr

The `-annotate-loop-instr` switch directs the compiler to provide additional annotation information for the prolog, kernel and epilog of a loop. See “[Assembly Optimizer Annotations](#)” [on page 2-80](#) for more details on this feature. Also, see also the `-no-annotate-loop-instr` switch ([on page 1-43](#)).

-auto-attrs

The `-auto-attrs` (automatic attributes) switch directs the compiler to emit automatic attributes based on the files it compiles. Emission of automatic attributes is enabled by default. See [“File Attributes” on page 1-348](#) for more information about attributes, and what automatic attributes the compiler emits. See also the `-no-auto-attrs` switch ([on page 1-43](#)) and the `-file-attr` switch ([on page 1-33](#)).

-build-lib

The `-build-lib` (build library) switch directs the compiler to use `elfar` (the librarian) to produce a library file (`.dlb`) as the output instead of using the linker to produce an executable file (`.dxe`). The `-o` option must be used to specify the name of the resulting library.

-C

The `-C` (comments) switch, which may only be run in combination with the `-E` or `-P` switches, directs the C/C++ preprocessor to retain comments in its output file.

-c

The `-c` (compile only) switch directs the compiler to compile and/or assemble the source files, but stop before linking. The output is an object file (`.doj`) for each source file.

-compatible-pm-dm

The `compatible-pm-dm` switch specifies that the compiler shall treat `dm`- and `pm`-qualified pointers as assignment-compatible.

-const-read-write

The `-const-read-write` switch directs the compiler to specify that constants may be accessed as read-write data (as in ANSI C). The compiler's default behavior assumes that data referenced through `const` pointers never changes.

The `-const-read-write` switch changes the compiler's behavior to match the ANSI C assumption, which is that other `non-const` pointers may be used to change the data at some point.



Invoke this switch with the **Pointers to const may point to non-const data** check box located in the Constants area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **Language Settings** category.

-const-strings

The `-const-strings` (`const-qualify strings`) switch directs the compiler to mark string literals as `const`-qualified. This is the default behavior. See also the `-no-const-strings` switch ([on page 1-44](#)).



Invoke this switch with the **Literal strings are const** check box located in the Constants area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **Language Settings** category.

-D *macro*[=*definition*]

The `-D` (define macro) switch directs the compiler to define a macro. If you do not include the optional definition string, the compiler defines the macro as the string '1'. Note that the compiler processes all `-D` switches on the command line before any `-U` (undefine macro) switches. For more information, see [“-U macro” on page 1-65](#).

-debug-types

The `-debug-types` switch builds a `*.h` file directly and writes a complete set of debugging information for the header file. The `-g` option need not be specified with the `-debug-types` switch because it is implied. For example,

```
cc21k -debug-types anyHeader.h
```

Until the introduction of `-debug-types`, the compiler would not accept an `*.h` file as a valid input file. The implicit `-g` option writes debugging information for only those typedefs that are referenced in the program. The `-debug-types` option provides complete debugging information for all typedefs and structs.

-double-size[-32|-64]

The `-double-size-32` (double is 32 bits) and the `-double-size-64` (double is 64 bits) switches select the storage format that the compiler uses for type `double`. The default mode is `-double-size-32`.

The C/C++ type `double` poses a special problem for the compiler. The C and C++ languages default to `double` for floating-point constants and many floating-point calculations. If `double` has the customary size of 64 bits, many programs inadvertently use slow speed emulated 64-bit floating-point arithmetic, even when variables are declared consistently as `float`.

To avoid this problem, `cc21k` provides a mode in which `double` is the same size as `float`. This mode is enabled with the `-double-size-32` switch and is the default mode.

Representing `double` using 32 bits gives good performance and provides enough precision for most DSP applications. This, however, does not fully conform to the C and C++ standards. The standard requires that `double`

Compiler Command-Line Interface

maintains 10 digits of precision, which requires 64 bits of storage. The `-double-size-64` switch sets the size of `double` to 64 bits for full standard conformance.

With `-double-size-32`, a `double` is stored in 32-bit IEEE single-precision format and is operated on using fast hardware floating-point instructions. Standard math functions such as `sin` also operate on 32-bit values. This mode is the default and is recommended for most programs. Calculations that need higher precision can be done with the `long double` type, which is always 64 bits.

With `-double-size-64`, a `double` is stored in 64-bit IEEE single-precision format and is operated on using slow floating-point emulation software. Standard math functions such as `sin` also operate on 64-bit values and are similarly slow. This mode is recommended only for porting code that requires that `double` have more than 32 bits of precision.

The `-double-size-32` switch defines the `__DOUBLES_ARE_FLOATS__` macro, while the `-double-size-64` switch undefines the `__DOUBLES_ARE_FLOATS__` macro.



Invoke this switch with the **Double size** radio buttons located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Compile** category, **Processor (1)** subcategory.

-double-size-any

The `-double-size-any` switch specifies that the resulting object files should be marked in such a way that will enable them to be linked against objects built with `doubles` either 32-bit or 64-bit in size.



Invoke this switch with the **Allow mixing of sizes** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Processor (1)** category.

-dry

The `-dry` (verbose dry run) switch directs the compiler to display main `cc21k` actions, but not to perform them.

-dryrun

The `-dryrun` (terse dry run) switch directs the compiler to display top-level `cc21k` actions, but not to perform them.

-E

The `-E` (stop after preprocessing) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling). The output (preprocessed source code) prints to the standard output stream unless the output file is specified with the `-o` switch. Note that the `-C` switch can only be run in combination with the `-E` switch.



Invoke this switch with the **Stop after: Preprocessor** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-ED

The `-ED` (run after preprocessing to file) switch directs the compiler to write the output of the C/C++ preprocessor to a file named `original_filename.i`. After preprocessing, compilation proceeds normally.



Invoke this switch with the **Generate preprocessed file** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-EE

The `-EE` (run after preprocessing) switch is similar to the `-E` switch, but it does not halt compilation after preprocessing.

-eh

The `-eh` (enable exception handling) switch directs the compiler to allow C++ code that contains catch statements and throw expressions and other features associated with ANSI/ISO standard C++ exceptions. When this switch is enabled, the compiler defines the macro `__EXCEPTIONS` to be 1.

If used when compiling C programs, without the `-c++` (C++ Mode) switch (on page 1-22), the `-eh` switch directs the compiler to generate exceptions tables but does not change the language accepted. In this case `__EXCEPTIONS` is not defined.

The `-eh` switch also causes the compiler to define `__ADI_LIBEH__` during the linking stage so that appropriate sections can be activated in the `.ldf` file, and the program can be linked with a library built with exceptions enabled.

Object files created with exceptions enabled may be linked with objects created without exceptions. However, exceptions can only be thrown from and caught, and cleanup code executed, in modules compiled with `-eh`. If an attempt is made to throw an exception through the execution of a function not compiled `-eh` then `abort` or the function registered with `set_terminate` is called. See also “[#pragma generate_exceptions_tables](#)” on page 1-218 and the `-no-eh` switch (on page 1-45).

In non-threaded applications, the buffer used for the passing of exception data is not returned to the heap on application exit. This is to avoid unnecessary code and will have no impact on behaviour.



Invoke this switch with the **C++ exceptions and RTTI** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Language Settings** category.

-enum-is-int

The `-enum-is-int` switch ensures that the type of an `enum` is `int`. By default, the compiler defines enumeration types with integral types larger than `int`, if `int` is insufficient to represent all the values in the enumeration. This switch prevents the compiler from selecting a type wider than `int`.

-extra-keywords

The `-extra-keywords` (enable short-form keywords) switch directs the compiler to recognize the Analog Devices keyword extensions to ANSI/ISO standard C and C++, such as `pm` and `dm`, without leading underscores, which can affect conforming ANSI/ISO C and C++ programs. This is the default mode.

The `-no-extra-keywords` switch ([on page 1-45](#)) can be used to disallow support for the additional keywords. [Table 1-15 on page 1-106](#) provides a list and a brief description of keyword extensions.

-file-attr *name*[=*value*]

The `-file-attr` (file attribute) switch directs the compiler to add the specified attribute name/value pair to all the files it compiles. To add multiple attributes, use the switch multiple times. If “=*value*” is omitted, the default value of “1” will be used. See the section “[File Attributes](#)” on [page 1-348](#) for more information about attributes, and what automatic attributes the compiler emits. See also the `-auto-attrs` switch ([on page 1-27](#)) and the `-no-auto-attrs` switch ([on page 1-43](#)).



Invoke this switch with the **Additional attributes** text field located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

Compiler Command-Line Interface

-flags-*{asm|compiler|lib|link|mem} switch [,switch2 [...]]*

The `-flags` (command-line input) switch directs the compiler to pass command-line switches to the other build tools.

The tools are listed in [Table 1-8](#):

Table 1-8. Switches Passed to Other Build Tools

Option	Tool
<code>-flags-asm</code>	Assembler
<code>-flags-compiler</code>	Compiler executable
<code>-flags-lib</code>	Library Builder (<code>elfar.exe</code>)
<code>-flags-link</code>	Linker
<code>-flags-mem</code>	Memory Initializer

-float-to-int

The `-float-to-int` switch instructs the compiler to use a support library function to convert a `float` to an integer. The library support routine performs extra checking to avoid a floating-point underflow occurring.

-force-circbuf

The `-force-circbuf` (circular buffer) switch instructs the compiler to make use of circular buffer facilities, even if the compiler cannot verify that the circular index or pointer is always within the range of the buffer. Without this switch, the compiler's default behavior is conservative, and does not use circular buffers unless it can verify that the circular index or pointer is always within the circular buffer range. See [“Circular Buffer Built-In Functions” on page 1-151](#).



Invoke this switch with the **Even when pointer may be outside buffer range** check box located in the VisualDSP++ Project Options dialog box, **Compile** page, **Language Settings** category.

-fp-associative

The `-fp-associative` switch directs the compiler to treat floating-point multiplication and addition as an associative. This switch is on by default. See the `-no-fp-associative` switch ([on page 1-45](#)) for more information.


-full-version


The `-full-version` (display versions) switch directs the compiler to display version information for build tools used in a compilation.

-g

The `-g` (generate debug information) switch directs the compiler to output symbols and other information used by the debugger.

When the `-g` switch is used in conjunction with the enable optimization (`-O`) switch, the compiler performs standard optimizations. The compiler also outputs symbols and other information to provide limited source-level debugging through the VisualDSP++ IDDE (debugger). This combination of options provides line debugging and global variable debugging.

 When the `-g` and `-O` switches are specified, no debug information is available for local variables and the standard optimizations can sometimes rearrange program code in a way that inaccurate line number information may be produced. For full debugging capabilities, use the `-g` switch without the `-O` switch. See also the `-Og` switch ([on page 1-50](#)).

 Invoke this switch by selecting the **Generate debug information** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-glite

The `-glite` (lightweight debugging) switch can be used on its own, or in conjunction with any of the `-g`, `-Og` or `-debug-types` compiler switches. When this switch is enabled it instructs the compiler to remove any unnecessary debug information for the code that is compiled.

When used on its own, the switch also enables the `-g` option.



This switch can be used to reduce the size of object and executable files, but will have no effect on the size of the code loaded onto the target.

-H

The `-H` (list headers) switch directs the compiler to output only a list of the files included by the preprocessor via the `#include` directive, without compiling.

-HH


The `-HH` (list headers and compile) switch directs the compiler to output to the standard output stream a list of the files included by the preprocessor via the `#include` directive. After preprocessing, compilation proceeds normally.

-h[elp]

The `-help` (command-line help) switch directs the compiler to output a list of command-line switches with a brief syntax description.

-I *directory* [{,|;} *directory*...]


The `-I` (include search directory) switch directs the C/C++ compiler preprocessor to append the directory (directories) to the search path for `include` files. This option can be specified more than once; all specified directories are added to the search path.

-  Invoke this switch with the **Additional include directories** text field located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Preprocessor** category.

Include files, whose names are not absolute path names and that are enclosed in “...” when included, are searched for in the following directories in this order:

1. The directory containing the current input file (the primary source file or the file containing the `#include`)
2. Any directories specified with the `-I` switch in the order they are listed on the command line
3. Any directories on the standard list:


`<VDSP++ install dir>/.../include`

-  If a file is included using the `<...>` form, this file is only searched for by using directories defined in items 2 and 3 above.

-I-

The `-I-` (start include directory list) switch establishes the point in the `include` directory list at which the search for header files enclosed in angle brackets begins. Normally, for header files enclosed in double quotes, the compiler searches in the directory containing the current input file; then the compiler reverts back to looking in the directories specified with the `-I` switch and then in the standard include directory.

It is possible to replace the initial search (within the directory containing the current input file) by placing the `-I-` switch at the point on the command line where the search for all types of header file begins. All `include` directories on the command line specified before the `-I-` switch are used only in the search for header files that are enclosed in double quotes.

-  The `-I` switch removes the directory containing the current input file from the `include` directory list.

Compiler Command-Line Interface

-i

The `-i` (less includes) switch can be used with the `-H`, `-HH`, `-M`, or `-MM` switches to direct the compiler to only output header details (`-H`, `-HH`) or makefile dependencies (`-M`, `-MM`) for `include` files specified in double quotes.

-implicit-pointers

The `-implicit-pointers` (implicit pointer conversion) switch allows a pointer to one type to be converted to a pointer to another without the use of an explicit cast. The compiler produces a discretionary warning rather than an error in such circumstances. This option is not valid when compiling in C++ mode.

For example, the following code will not compile without this switch:

```
int *foo(int *a) {
    return a;
}
int main(void) {
    char *p = 0, *r;
    r = foo(p);          /* Bad: normally produces an error */
    return 0;
}
```

In this example, both the argument to `foo` and the assignment to `r` will be faulted by the compiler. Using `-implicit-pointers` converts these errors into warnings.



Invoke the `-implicit-pointers` switch with the **Allow incompatible pointer types** check box located in the VisualDSP++ Project Options dialog box, **Compile** page, **Language Settings** category.

-include *filename*

The `-include` (include file) switch directs the preprocessor to process the specified file before processing the regular input file. Any `-D` and `-U` options on the command line are always processed before an `-include` file. Only one `-include` may be given.

-ipa

The `-ipa` (interprocedural analysis) switch turns on Interprocedural Analysis (IPA) in the compiler. This option enables optimization across the entire program, including between source files that were compiled separately. If used, the `-ipa` option should be applied to all C and C++ files in the program. [For more information, see “Interprocedural Analysis” on page 1-84.](#) Specifying `-ipa` also implies setting the `-O` switch ([on page 1-49](#)).



Invoke this switch by selecting the **Interprocedural Analysis** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-L *directory*[{;|,} *directory*...]

The `-L` (library search directory) switch directs the compiler to append the directory to the search path for library files.

-l *library*

The `-l` (link library) switch directs the compiler to search the library for functions and global variables when linking. The library name is the portion of the file name between the `lib` prefix and `.dll` extension.

For example, the `-lc` compiler switch directs the linker to search in the library named `c`. This library resides in a file named `libc.dll`.

All object files should be listed on the command line before listing libraries using the `-l` switch. When a reference to a symbol is made, the symbol definition will be taken from the left-most object or library on the com-

Compiler Command-Line Interface

mand line that contains the global definition of that symbol. If two objects on the command line contain definitions of the symbol x , x will be taken from the left-most object on the command line that contains a global definition of x .

If one of the definitions for x comes from user objects, and the other from a user library, and the library definition should be overridden by the user object definition, it is important that the user object comes before the library on the command line.

Libraries included in the default `.ldf` file are searched last for symbol definitions.

-list-workarounds

The `-list-workarounds` (list supported errata workarounds) switch displays a list of all errata workarounds which the compiler supports. See [“Controlling Silicon Revision and Anomaly Workarounds within the Compiler” on page 1-85](#) for details of valid workarounds and the interaction of the `-si-revision`, `-workaround` and `-no-workaround` switches

-M

The `-M` (generate make rules only) switch directs the compiler not to compile the source file, but to output a rule suitable for the make utility, describing the dependencies of the main program file. The format of the make rule output by the preprocessor is:

```
object-file: include-file ...
```

-MD

The `-MD` (generate make rules and compile) switch directs the preprocessor to print to a file called `original_filename.d` a rule describing the dependencies of the main program file. After preprocessing, compilation proceeds normally. See also the `-Mo` switch.

-MM

The `-MM` (generate make rules and compile) switch directs the preprocessor to print to standard out a rule describing the dependencies of the main program file. After preprocessing, compilation proceeds normally.

-Mo *filename*

The `-Mo filename` (preprocessor output file) switch directs the compiler to use *filename* for the output of `-MD` or `-ED` switches.

-Mt *name*

The `-Mt name` (output make rule for the named source) switch modifies the target of generated dependencies, renaming the target to *name*. It only has an effect when used in conjunction with the `-M` or `-MM` switch.

-map *filename*

The `-map filename` (generate a memory map) switch directs the compiler to output a memory map of all symbols. The map file name corresponds to the *filename* argument. For example, if the argument is `test`, the map file name is `test.xml`. The `.xml` extension is added where necessary.

-mem

The `-mem` (enable memory initialization) switch directs the compiler to run the `mem21k` initializer (utility). The memory initializer can be controlled through the `-flags-mem` switch ([on page 1-34](#)). See the `-no-mem` switch ([on page 1-45](#)) for more information.

-multiline

The `-multiline` switch enables a compiler GNU compatibility mode which allows string literals to span multiple lines without the need for a “\” at the end of each line. This is the default mode. See the `-no-multiline` switch ([on page 1-46](#)) for more information.



Invoke this switch with the **Allow multi-line character strings** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Language Settings** category.

-never-inline

The `-never-inline` switch instructs the compiler to ignore the `inline` qualifier on function definitions, so that no calls to such functions will be inlined. See also [“-always-inline” on page 1-26](#).



Invoke this switch with the **Never** check box located in the Inlining area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-no-aligned-stack

The `-no-aligned-stack` (disable stack alignment) switch directs the compiler to not align the program stack on a double-word boundary. For more information, see [“-aligned-stack” on page 1-25](#).

-no-alttok

The `-no-alttok` (disable alternative tokens) switch directs the compiler not to accept alternative operator keywords and digraph sequences in the source files. This is the default mode. For more information, see [“-alttok” on page 1-25](#).

-no-annotate

The `-no-annotate` (disable assembly annotations) switch directs the compiler not to annotate assembly files generated by the compiler. The default behavior is that whenever optimizations are enabled all assembly files generated by the compiler are annotated with information on the

performance of the generated assembly. See [“Assembly Optimizer Annotations” on page 2-80](#) for more details on this feature. For more information, see [“-annotate” on page 1-26](#).



Invoke this switch by clearing the **Generate assembly code annotations** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-no-annotate-loop-instr

The `-no-annotate-loop-instr` switch disables the production of additional loop annotation information by the compiler. This is the default mode. See the `-annotate-loop-instr` switch ([on page 1-26](#)).

-no-auto-attrs

The `-no-auto-attrs` (no automatic attributes) switch directs the compiler not to emit automatic attributes based on the files it compiles. Emission of automatic attributes is enabled by default. See [“File Attributes” on page 1-348](#) for more information about attributes, and what automatic attributes the compiler emits. See also the `-auto-attrs` switch ([on page 1-27](#)) and the `-file-attr` switch ([on page 1-33](#)). For more information, see [“-auto-attrs” on page 1-27](#).



Invoke this switch by clearing the **Auto-generated attributes** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-no-builtin

The `-no-builtin` (no built-in functions) switch directs the compiler not to generate short names for the built-in functions (for example, `abs()`), and to accept only the full name (for example, `__builtin_abs()`). Note that this switch influences many functions. This switch also predefines the `__NO_BUILTIN` preprocessor macro. For more information on built-in functions, see [“C++ Fractional Type Support” on page 1-228](#).



Invoke this switch by selecting the **Disable builtin functions** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Language Settings** category.

-no-circbuf

The `-no-circbuf` (no circular buffer) switch disables the automatic generation of circular buffer code by the compiler. Uses of the `circindex()` and `circptr()` functions (that is, explicit circular buffer operations) are not affected.



Invoke this switch with the **Never** check box located in the Circular Buffer Generation area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **Language Settings** category.

-no-const-strings

The `-no-const-strings` switch directs the compiler not to make string literals `const` qualified. See the `-const-strings` switch ([on page 1-28](#)) for more information.

-no-db

The `-no-db` (no delayed branches) switch specifies that the compiler shall not generate jumps that use delayed branches.



Disabling of interrupts within the epilogue code of a re-entrant interrupt function still uses a delayed branch jump to minimise interrupt latency.

-no-defs

The `-no-defs` (disable defaults) switch directs the preprocessor not to define any default preprocessor macros, include directories, library directories, libraries, and run-time headers. It also disables the Analog Devices cc21k C/C++ keyword extensions.

-no-eh

The `-no-eh` (disable exception handling) switch directs the compiler to disallow ANSI/ISO C++ exception handling. This is the default mode. See the `-eh` switch ([on page 1-32](#)) for more information.

-no-extra-keywords

The `-no-extra-keywords` (disable short-form keywords) switch directs the compiler not to recognize the Analog Devices keyword extensions that might affect conformance to ISO/ANSI standards for C and C++ languages. These include keywords such as `pm` and `dm`, which may be used as identifiers in standard conforming programs. Alternate keywords, which are prefixed with two leading underscores, such as `__pm` and `__dm`, continue to work. See the `-extra-keywords` switch ([on page 1-33](#)) for more information.



Invoke this switch with the **Disable Analog Devices extension keywords** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Language Settings** category.

-no-fp-associative

The `-no-fp-associative` switch directs the compiler NOT to treat floating-point multiplication and addition as an associative. See the `-fp-associative` switch ([on page 1-35](#)) for more information.



Invoke this switch with the **Do not treat floating point operations as associative** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Language Settings** category.

-no-mem

The `-no-mem` (disable memory initialization) switch directs the compiler not to run the `mem21k` initializer. Note that if you use `-no-mem`, the compiler does not initialize globals and statics. See the `-mem` switch ([on page 1-41](#)) for more information.

-no-multiline

The `-no-multiline` switch disables a compiler GNU compatibility mode which allows string literals to span multiple lines without the need for a “\” at the end of each line. See the `-multiline` switch ([on page 1-41](#)) for more information.



Invoke this switch by clearing the **Allow multi-line character strings** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Language Settings** category.

-no-progress-rep-timeout

The `-no-progress-rep-timeout` (disable progress message for long compilations) switch disables the diagnostic message issued by the compiler to indicate that it is still working, when a function’s compilation is taking an excessively long time. The message is disabled by default. See also the `-progress-rep-timeout` switch ([on page 1-58](#)) and the `-progress-rep-timeout-secs` switch ([on page 1-58](#)).

-no-sat-associative

The `-no-sat-associative` (saturating addition is not associative) switch instructs the compiler not to consider saturating addition operations as associative: $(a+b)+c$ may not be rewritten as $a+(b+c)$, when the addition operator saturates. The default is that saturating addition is not associative. See the `-sat-associative` switch ([on page 1-60](#)) for more information.


-no-saturation

The `-no-saturation` switch directs the compiler not to introduce faster operations in cases where the faster operation would saturate (if the expression overflowed) when the original operation would have wrapped the result. The code produced may be less efficient than when the switch

is not used. Saturation is enabled by default when optimizing, and may be disabled by this switch. Saturation is disabled when not optimizing (this switch is the default when not optimizing).


-no-shift-to-add

The `-no-shift-to-add` switch prevents the compiler from replacing a shift-by-one instruction with an addition. While this can produce faster code, it can also lead to arithmetic overflow.

 Invoke this switch from the **Disable shift-to-add conversion** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Processor (1)** category.

-no-simd

The `-no-simd` (disable SIMD mode) switch directs the compiler to disable automatic SIMD code generation when compiling for ADSP-2116x, ADSP-2126x and ADSP-213xx processors. Note that SIMD code is still generated for a loop if it is preceded with the “`SIMD_for`” pragma. The pragma is treated as an explicit user request to generate SIMD code and is always obeyed, if possible. See [“SIMD Support” on page 1-231](#) for more information.

 Invoke this switch from the **Disable automatic SIMD code generation** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Processor (1)** category.

-no-std-ass

The `-no-std-ass` (disable standard assertions) switch prevents the compiler from defining the standard assertions. See the `-A` switch ([on page 1-23](#)) for the list of standard assertions.

Compiler Command-Line Interface

-no-std-def

The `-no-std-def` (disable standard macro definitions) switch prevents the compiler from defining default preprocessor macro definitions.



This switch also disables the Analog Devices keyword extensions that have no leading underscores, such as `pm` and `dm`.

-no-std-inc

The `-no-std-inc` (disable standard include search) switch directs the C/C++ preprocessor to search for header files in the current directory and directories specified with the `-I` switch.



You can invoke this switch by selecting the **Ignore standard include paths** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-no-std-lib

The `-no-std-lib` (disable standard library search) switch directs the compiler to search for libraries in only the current project directory and directories specified with the `-L` switch.

-no-threads

The `-no-threads` (disable thread-safe build) switch specifies that all compiled code and libraries used in the build need not be thread-safe. This is the default setting when the `-threads` (enable thread-safe build) switch is not used.

-no-workaround *workaround_id[,workaround_id ...]*

The `-no-workaround` *workaround_id* (disable avoidance of specific errata) switch disables compiler code generator workarounds for specific hardware errata. See [“Controlling Silicon Revision and Anomaly Workarounds](#)

[within the Compiler” on page 1-85](#) for details of valid workarounds and the interactions of the `-si-revision`, `-workaround` and `-no-workaround` switches.

-normal-word-code

The `-normal-word-code` switch has the same effect as compiling with the `-nwc` switch. It directs the compiler to generate instructions of normal word size (48-bits). This switch applies only when compiling code targeted for 2146x processors.

-nwc

The `-nwc` switch has the same effect as compiling with the `-normal-word-code` switch. It directs the compiler to generate instructions of normal word size (48-bits). This switch applies only when compiling code targeted for 2146x processors.

-O[0 | 1]

The `-O` (enable optimizations) switch directs the compiler to produce code that is optimized for performance. Optimizations are not enabled by default for the `cc21k` compiler. (Note that the switch settings are numbers—zeros or 1s—while the switch itself is the letter “O.”) The switch setting `-O` or `-O1` turns optimization on, while setting `-O0` turns off all optimizations.



You can invoke this switch by selecting the **Enable optimization** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category.

-Oa

The `-Oa` (automatic function inlining) switch enables the inline expansion of C/C++ functions, which are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the `-Ov` (optimize for speed versus size) switch ([on page 1-50](#)).

Compiler Command-Line Interface

Therefore, use of `-Ov100` indicates that as many functions as possible are auto-inlined, whereas `-Ov0` prevents any function from being auto-inlined. Specifying `-Oa` also implies the use of `-O`.



Invoke this switch with the **Automatic** check box located in the Inlining area of the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-Og

The `-Og` switch enables a compiler mode that attempts to perform optimizations while still preserving the debugging information. It is meant as an alternative for those who want a debuggable program but who are also concerned about the performance of their debuggable code and are less concerned about the compilation time.

-Os

The `-Os` (optimize for size) switch directs the compiler to produce code that is optimized for size. This is achieved by performing all optimizations except those that increase code size. The optimizations not performed include loop unrolling, some delay slot filling, and jump avoidance.

-Ov *num*

The `-Ov num` (optimize for speed versus size) switch informs the compiler of the relative importance of speed versus size, when considering whether such trade-offs are worthwhile. The *num* variable should be an integer between 0 (purely size) and 100 (purely speed).

For any given optimization, the compiler modifies the code being generated. Some optimizations produce code that will execute in fewer cycles, but which will require more code space. In such cases, there is a trade-off between speed and space.

The *num* variable indicates a sliding scale between 0 and 100 which is the probability that a linear piece of generated code—a “basic block”—will be optimized for speed or for space. At `-Ov0` all blocks are optimized for space and at `-Ov100` all blocks are optimized for speed. At any point in between, the decision is based upon *num* and how many times the block is expected to be executed—the “execution count” of the block. Figure 1-1 demonstrates this relationship.

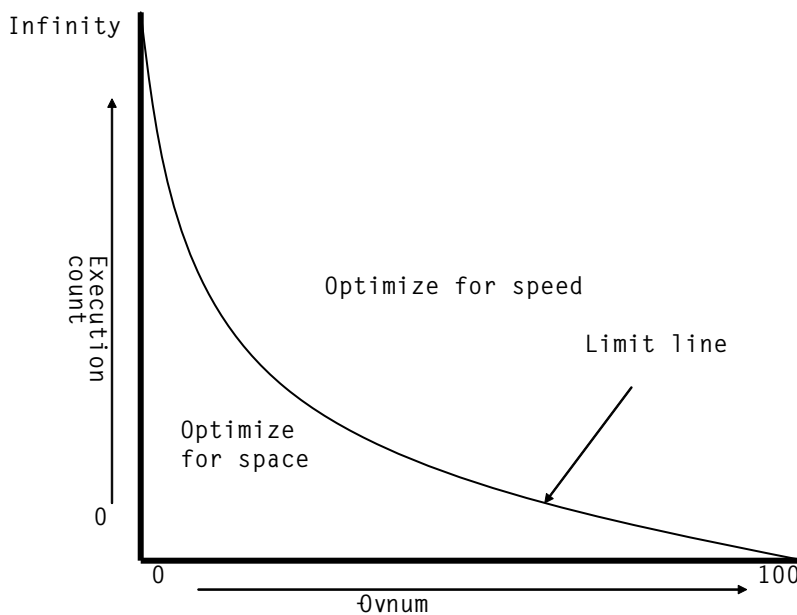


Figure 1-1. `-Ov` Switch Optimization Curve

For any given optimization where speed and size conflict, the potential benefit is dependent on the execution count: an optimization that increases performance at the expense of code size is considerably more beneficial if applied to the core loop of a critical algorithm than if applied

to one-time initialization code or to rarely-used error-handling functions. If code appears to be executed only once, it will be optimized for space. As its execution count increases, so too does the likelihood that the compiler will consider the code increase worthwhile for the corresponding benefit in performance.

As [Figure 1-1](#) shows, the `-Ov` switch affects the point at which a given execution count is considered sufficient to switch optimization from “for space” to “for speed”. Where *num* is a low value, the compiler is biased towards space, so a block’s execution count has to be relatively high for the compiler to apply code-increasing transformations. Where *num* has a high value, the compiler is biased towards speed, so the same transformation will be considered valid for a much lower execution count.

The `-Ov` switch is most effective when used in conjunction with profile-guided optimization, where accurate execution counts are available. Without profile-guided optimization, the compiler makes estimates of the relative execution counts using heuristics.



Invoke this switch with the **Optimize for code size/speed** slider located in the **VisualDSP++ Project Options** dialog box, **Compile** page, **General** category.

[For more information, see “Using Profile-Guided Optimization” in Chapter 2, Achieving Optimal Performance from C/C++ Source Code.](#)

-o *filename*

The `-o` (output file) switch directs the compiler to use *filename* for the name of the final output file.

-overlay

The `-overlay` (program may use overlays) switch will disable the propagation of register information between functions and force the compiler to assume that all functions clobber all scratch registers. Note that this switch will affect all functions in the source file, and may result in a performance

degradation. For information on disabling the propagation of register information only for specific functions, see “[#pragma overlay](#)” on [page 1-183](#).

-overlay-clobbers *clobbered-regs*

The `-overlay-clobbers` (registers clobbered by overlay manager) switch identifies the set of registers clobbered by an overlay manager, if one is used. The compiler will assume that any call to an overlay-managed function will clobber the values in `clobbered-regs`, in addition to those clobbered by the function in question. A function is considered to be an overlay-managed function if the `-overlay` switch ([on page 1-52](#)) is specified, or if the function is marked with `#pragma overlay` ([on page 1-183](#)).

The *clobbered-regs* variable is a single string formatted as per the argument to `#pragma regs_clobbered`, except that individual components of the list may also be separated by commas.



Whitespace and semi-colons are valid separators for the components of the list, but must be properly quoted when being passed to the compiler.

Examples:

```
cc21k -O t.c -overlay -overlay-clobbers r3,m4,r5
cc21k -O t.c -overlay -overlay-clobbers Dscratch
cc21k -O t.c -overlay -overlay-clobbers "r3 m4;r5"
```

-P

The `-P` (omit line numbers) switch directs the compiler to stop after the C/C++ preprocessor runs (without compiling) and to omit the `#line` preprocessor command with line number information from the preprocessor output. The `-C` switch can be used in conjunction with `-P` to retain comments.

Compiler Command-Line Interface

-PP

The `-PP` (omit line numbers and compile) switch is similar to the `-P` switch; however, it does not halt compilation after preprocessing.

-path-{ asm | compiler | lib | link } *pathname*

The `-path-{asm|compiler|lib|link} pathname` (tool location) switch directs the compiler to use the specified component in place of the default-installed version of the compilation tool. The component comprises a relative or absolute path to its location. Respectively, the tools are the assembler, compiler, librarian, or linker. Use this switch when overriding the normal version of one or more of the tools. The `-path-{...}` switch also overrides the directory specified by the `-path-install` switch.

-path-install *directory*

The `-path-install` (installation location) switch directs the compiler to use the specified directory as the location for all compilation tools instead of the default path. This is useful when working with multiple versions of the tool set.



You can selectively override this switch with the `-path-{asm|compiler|lib|link}` switch.

-path-output *directory*

The `-path-output` (non-temporary files location) switch directs the compiler to place final output files in the specified directory.

-path-temp *directory*

The `-path-temp` (temporary files location) switch directs the compiler to place temporary files in the specified directory.

-pch

The `-pch` (precompiled header) switch directs the compiler to automatically generate and use precompiled header files. A precompiled output header has a `.pch` extension attached to the source file name. By default, all precompiled headers are stored in a directory called `PCHRepository`.

-pchdir *directory*

The `-pchdir` (locate `PCHRepository`) switch specifies the location of an alternative `PCHRepository` for storing and invocation of precompiled header files. If the directory does not exist, the compiler creates it. Note that `-o` (output) does not influence the `-pchdir` option.


-pgo-session *session-id*

The `-pgo-session` (specify PGO session identifier) switch is used with profile-guided optimization. It has the following effects:

- When used with the `-pguide` switch ([on page 1-56](#)), the compiler associates all counters for this module with the session identifier `session-id`.
- When used with a previously-gathered profile (a `.pgo` file), the compiler ignores the profile contents, unless they have the same `session-id` identifier.

This is most useful when the same source file is being built in more than one way (for example, different macro definitions, or for multiple processors) in the same application; each variant of the build can have a different `session-id` associated with it, which means that the compiler will be able to identify which parts of the gathered profile should be used when optimizing for the final build.


If each source file is only built in a single manner within the system (the usual case), then the `-pgo-session` switch is not needed.

 Invoke this switch with the **PGO session name** text field located in the **VisualDSP++ Project Options** dialog box, **Compile** page, **Profile-Guided Optimization** category.

For more information, see [“Using Profile-Guided Optimization” in Chapter 2, Achieving Optimal Performance from C/C++ Source Code.](#)

-pguide

The `-pguide` switch causes the compiler to add instrumentation for the gathering of a profile (a `.pgo` file) as the first stage of performing profile-guided optimization.

 Invoke this switch with the **Prepare application to create new profile** check box located in the **VisualDSP++ Project Options** dialog box, **Compile** page, **Profile-Guided Optimization** category.

For more information, see [“Using Profile-Guided Optimization” in Chapter 2, Achieving Optimal Performance from C/C++ Source Code.](#)

-pplist *filename*

The `-pplist` (preprocessor listing) directs the preprocessor to output a listing to the named file. When more than one source file is preprocessed, the listing file contains information about the last file processed. The generated file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the compiler.

Each listing line begins with a key character that identifies its type as shown in [Table 1-9](#):

Table 1-9. Key Characters

Character	Meaning
N	Normal line of source
X	Expanded line of source
S	Line of source skipped by <code>#if</code> or <code>#ifdef</code>
L	Change in source position
R	Diagnostic message (remark)
W	Diagnostic message (warning)
E	Diagnostic message (error)
C	Diagnostic message (catastrophic error)

-proc processor

The `-proc processor` (target processor) switch specifies the compiler produces code suitable for the specified processor. Refer to VisualDSP++ online Help for the list of supported SHARC processors. For example,

```
cc21k -proc ADSP-21161 -o bin\p1.doj p1.asm
```



If no target is specified with the `-proc` switch, the system uses the ADSP-21060 processor settings as a default.

When compiling with the `-proc` switch, the appropriate processor macro as well as `__ADSP21000__` are defined as 1. For example, `__ADSP21060__` and `__ADSP-21000__` are 1.



See also “[-si-revision version](#)” on page 1-62 for more information on silicon revision of the specified processor.

-progress-rep-func

The `-progress-rep-func` switch provides feedback on the compiler’s progress that may be useful when compiling and optimizing very large source files. It issues a “warning” message each time the compiler starts

Compiler Command-Line Interface

compiling a new function. The “warning” message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1472`.

-progress-rep-opt

The `-progress-rep-opt` switch provides feedback on the compiler’s progress that may be useful when compiling and optimizing a very large, complex function. It issues a “warning” message each time the compiler starts a new optimization pass on the current function. The “warning” message is a remark that is disabled by default, and this switch enables the remark as a warning. The switch is equivalent to `-Wwarn=cc1473`.

-progress-rep-timeout

The `-progress-rep-timeout` switch issues a diagnostic message if the compiler exceeds a time limit during compilation. This indicates the compiler is still operating, just taking a long time.


-progress-rep-timeout-secs secs

The `-progress-rep-timeout-secs` switch specifies how many seconds must elapse during a compilation before the compiler issues a diagnostic message about the length of time the compilation has used so far.

-R *directory* [{:|,}*directory* ...]


The `-R directory` (add source directory) switch directs the compiler to add the specified directory to the list of directories searched for source files. On Windows platforms, multiple source directories are given as a colon, comma, or semicolon separated list.

The compiler searches for the source files in the order specified on the command line. The compiler searches the specified directories before reverting to the current project directory. The `-R directory` option is position-dependent on the command line. That is, it affects only source files that follow the option.

 Source files whose file names begin with `/`, `./` or `../` (or Windows equivalent) and contain drive specifiers (on Windows platforms) are not affected by this option

-R-

The `-R-` (disable source path) switch removes all directories from the standard search path for source files, effectively disabling this feature.

 This option is position-dependent on the command line; it only affects files following it.

-reserve *register* [, *register* ...]

The `-reserve` (reserve register) switch directs the compiler not to use the specified registers. This guarantees that a known set of registers are available for inline assembly code or linked assembly modules. Separate each register name with a comma on the compiler command line.

You can reserve the following registers: `b0`, `l0`, `m0`, `i0`, `b1`, `l1`, `m1`, `i1`, `b8`, `l8`, `m8`, `i8`, `b9`, `l9`, `m9`, `i9`, `ustat1`, and `ustat2` (as well as `ustat3` and `ustat4` on ADSP-2116x, ADSP-2126x, ADSP-2136x processors and ADSP-2137x processors). When reserving an `L` (length) register, you must reserve the corresponding `I` (index) register; reserving an `L` register without reserving the corresponding `I` register may result in execution problems.

-restrict-hardware-loops *maximum*

The `-restrict-hardware-loops maximum` switch restricts the level of nested hardware loops that the compiler generates. The default setting is 6, which is the maximum number of levels that the hardware supports.

Compiler Command-Line Interface

-sat-associative

The `-sat-associative` (saturating addition is associative) switch instructs the compiler to consider saturating addition operations as associative: $(a+b)+c$ may be rewritten as $a+(b+c)$, when the addition operator saturates. The default is that saturating addition is not associative.

-S

The `-S` (stop after compilation) switch directs `cc21k` to stop compilation before running the assembler. The compiler outputs an assembly file with an `.s` extension.

-s

The `-s` (strip debug information) switch directs the compiler to remove debug information (symbol table and other items) from the output executable file during linking.

-save-temps

The `-save-temps` (save intermediate files) switch directs the compiler to retain intermediate files, generated and normally removed as part of the various compilation stages. These intermediate files are placed in the `-path-output` specified output directory or the build directory if the `-path-output` switch is not used. See [Table 1-3 on page 1-10](#) for a list of intermediate files.



Invoke this switch with the **Save temporary files** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **General** category.

-section *id=section_name[, id=section_name...]*

The `-section` switch controls the placement of types of data produced by the compiler. The data is placed into the section “`section_name`” as specified on the command line.

The compiler currently supports the following section identifiers:

<code>code</code>	Controls placement of machine instructions Default is <code>seg_pmco</code> .
<code>data</code>	Controls placement of initialized variable data Default is <code>seg_dmda</code>
<code>pm_data</code>	Controls placement of initialized data declared with the <code>_pm</code> keyword
<code>constdata</code>	Controls placement of constant data
<code>pm_constdata</code>	Controls placement of constant data declared with the <code>_pm</code> keyword
<code>bsz</code>	Controls placement of zero-initialized variable data Default is <code>seg_dmda</code> .
<code>sti</code>	Controls placement of the static C++ class constructor “start” functions Default is <code>seg_pmco</code> . For more information, see “Constructors and Destructors of Global Class Instances” on page 1-280.
<code>switch</code>	Controls placement of jump-tables used to implement C/C++ switch statements.
<code>strings</code>	Controls placement of string literals.
<code>vtbl</code>	Controls placement of the C++ virtual lookup tables Default is <code>seg_vtbl</code> .
<code>vtable</code>	Synonym for <code>vtbl</code>
<code>autoinit</code>	Controls placement of data used to initialise aggregate autos
<code>alldata</code>	Controls placement of data, <code>constdata</code> , <code>bss</code> , <code>strings</code> and <code>autoinit</code> all at once

Please note that `alldata` is not a real section *kind*, but rather a placeholder for `data`, `constdata`, `bsz`, `strings` and `autoinit`. Therefore,

```
-section alldata=X
```

is equivalent to

```
-section data=X -section constdata=X -section bsz=X
-section strings=X -section autoinit=X
```

Compiler Command-Line Interface

Make sure that the section selected via the command line exists within the `.ldf` file. (Refer to the “Linker” chapter in the *VisualDSP++ 5.0 Linker and Utilities Manual*.)

-short-word-code

The `-short-word-code` switch has the same effect as compiling with the `-swc` switch [on page 1-63](#). It directs the compiler to generate instructions of short word size (16/32/48-bits). This switch only applies when compiling code targeted for 2146x processors and is the default setting.

-show

The `-show` (display command line) switch shows the command-line arguments passed to `cc21k`, including expanded option files and environment variables. This option allows you to ensure that command-line options have been passed successfully.

-si-revision *version*

The `-si-revision version` (silicon revision) switch directs the compiler to build for a specific hardware revision (version). Any errata workarounds available for the targetted silicon revision will be enabled. [For more information, see “Controlling Silicon Revision and Anomaly Workarounds within the Compiler” on page 1-85.](#)

-signed-bitfield

The `-signed-bitfield` (make plain bit-fields signed) switch directs the compiler to make bit-fields (which have not been declared with an explicit signed or unsigned keyword) to be signed. This switch does not affect plain one-bit bit-fields which are always unsigned. This is the default mode. See also the `-unsigned-bitfield` switch ([on page 1-65](#)).

-structs-do-not-overlap

The `-structs-do-not-overlap` switch specifies that the source code being compiled contains no structure copies such that the source and the destination memory regions overlap each other in a non-trivial way.

For example, in the statement

```
*p = *q;
```

where `p` and `q` are pointers to some structure type `S`, the compiler, by default, always ensures that, after the assignment, the structure pointed to by “`p`” contains an image of the structure pointed to by “`q`” prior to the assignment. In the case where `p` and `q` are not identical (in which case the assignment is trivial) but the structures pointed to by the two pointers may overlap each other, doing this means that the compiler must use the functionality of the C library function “`memmove`” rather than “`memcpy`”.

It is slower to use “`memmove`” to copy data than it is to use “`memcpy`”. Therefore, if your source code does not contain such overlapping structure copies, you can obtain higher performance by using the command-line switch `-structs-do-not-overlap`.



Invoke this switch from the Structs/classes do not overlap check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Language Settings** category.

-SWC

The `-swc` switch has the same effect as compiling with the `-short-word-code` switch [on page 1-62](#). It directs the compiler to generate instructions of short-word size (16/32/48-bits). This switch only applies when compiling code targeted for 2146x processors and is the default setting.

Compiler Command-Line Interface

-syntax-only

The `-syntax-only` (check syntax only) switch directs the compiler to check the source code for syntax errors but not to write any output.

-sysdefs

The `-sysdefs` (system definitions) switch directs the compiler to define several preprocessor macros describing the current user and user's system. The macros are defined as character string constants and are used in functions with null-terminated string arguments.

The following macros are defined if the system returns information for them ([Table 1-10](#)).

Table 1-10. System Macros Defined

Macro	Description
<code>__HOSTNAME__</code>	The name of the host machine
<code>__SYSTEM__</code>	The Operating System name of the host machine
<code>__USERNAME__</code>	The current user's login name


-T *filename*

The `-T` (linker description file) switch directs the compiler to use the specified linker description file (`.ldf`) as control input for linking. If `-T` is not specified, a default `.ldf` file is selected based on the processor variant.

-threads

When used, the `-threads` switch defines the macro `_ADI_THREADS` as one (1) at the compile, assemble and link phases of a build. This specifies that certain aspects of the build are to be done in a thread-safe way.

When applications are built within VisualDSP++, this switch is added automatically to projects that have VDK support selected.

 The use of thread-safe libraries is necessary in conjunction with the `-threads` flag when using the VisualDSP++ Kernel (VDK). The thread-safe libraries can be used with other RTOSs but this requires the definition of various VDK interfaces.


The use of the `-threads` switch does not imply that the compiler will produce thread-safe code when compiling C/C++ source. Make sure to use multi-threaded programming practises in your code (such as semaphores to access shared data).

-time

The `-time` (tell time) switch directs the compiler to display the elapsed time as part of the output information about each phase of the compilation process.

-U *macro*

The `-U` (undefine macro) switch directs the compiler to undefine macros. If you specify a macro name, it is undefined. Note that the compiler processes all `-D` (define macro) switches on the command line before any `-U` (undefine macro) switches. For more information, see [“-D macro\[=definition\]” on page 1-28](#).

 Invoke this switch by entering macro names to be undefined, separated by commas, in the **Undefines** field in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Preprocessor** category.

-unsigned-bitfield

The `-unsigned-bitfield` (make plain bit-fields unsigned) switch directs the compiler to make bit-fields which have not been declared with an explicit signed or unsigned keyword to be unsigned. This switch does not affect plain one-bit bit-fields which are always unsigned.

Compiler Command-Line Interface

For example, given the declaration

```
struct {  
    int a:2;  
    int b:1;  
    signed int c:2;  
    unsigned int d:2;  
} x;
```

[Table 1-11](#) lists the bit-field values.

Table 1-11. Bit-Field Values

Field	-unsigned-bitfield	-signed-bitfield	Why
x.a	-2..1	0..3	Plain field
x.b	0..1	0..1	One bit
x.c	-2..1	-2..1	Explicit signed
x.d	0..3	0..3	Explicit unsigned

See also the `-signed-bitfields` switch ([on page 1-62](#)).

-v

The `-v` (version and verbose) switch directs the compiler to display both the version and command-line information for all the compilation tools as they process each file.

-verbose

The `-verbose` (display command line) switch directs the compiler to display command-line information for all the compilation tools as they process each file.

-version

The `-version` (display version) switch directs the compiler to display version information for all the compilation tools as they process each file.

-W{error|remark|suppress|warn} number[,number ...]

The `-W{...} number` (override error message) switch directs the compiler to override the severity of the specified diagnostic messages (errors, remarks, or warnings). The *number* argument specifies the message to override.

At compilation time, the compiler produces a number for each specific compiler diagnostic message. The `{D}` (discretionary) string after the diagnostic message number indicates that the diagnostic may have its severity overridden. Each diagnostic message is identified by a number that is used across all compiler software releases.



If the processing of the compiler command line generates a diagnostic, the position of the `-W` switch on the command line is important. If the `-W` switch changes the severity of the diagnostic, it must occur before the command line switch that generates the diagnostic; otherwise, no change of severity will occur.

Also, as shown in the Output window and in Help, error codes sometimes begin with a leading zero (for example, `cc0025`). If you try to suppress error codes with `#pragma diag()` or `-W{error|remark|suppress|warn}`, and supply the code with a leading zero, it does not work. This is because the compiler reads the number as an octal value, and will suppress a different warning or error.

-Werror-limit number

The `-Werror-limit` (maximum compiler errors) switch lets you set a maximum number of errors for the compiler before it aborts.

-Werror-warnings

The `-Werror-warnings` (treat warnings as errors) switch directs the compiler to treat all warnings as errors, with the result that a warning will cause the compilation to fail.

Compiler Command-Line Interface

-Wremarks

The `-Wremarks` (enable diagnostic warnings) switch directs the compiler to issue remarks, which are diagnostic messages milder than warnings.



Invoke this switch by selecting the **Enable remarks** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Warning** selection.

-Wterse

The `-Wterse` (enable terse warnings) switch directs the compiler to issue the briefest form of warnings. This also applies to errors and remarks.

-w

The `-w` (disable all warnings) switch directs the compiler not to issue warnings.

Invoke this switch by selecting the **Disable all warnings and remarks** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **Warning** selection.



If the processing of the compiler command line generates a warning, the position of the `-w` switch on the command line is important. If the `-w` switch is located before the command line switch that causes the warning, the warning will be suppressed; otherwise, it will not be suppressed.

-warn-protos

The `-warn-protos` (warn if incomplete prototype) switch directs the compiler to issue a warning when it calls a function for which an incomplete function prototype has been supplied. This option has no effect in C++ mode.

Invoke this switch with the **Function declarations without prototypes** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Warning** category.

-workaround *workaround_id[,workaround_id ...]*

The `-workaround workaround_id [,workaround_id...]` (enable avoidance of specific errata) switch enables compiler code generator workarounds for specific hardware errata. See [“Controlling Silicon Revision and Anomaly Workarounds within the Compiler”](#) on page 1-85 for details of valid workarounds and the interaction of the `-si-revision`, `-workaround` and `-no-workaround` switches.

-write-files

The `-write-files` (enable driver I/O redirection) switch directs the compiler driver to redirect the file name portions of its command line through a temporary file. This technique helps to handle long file names, which can make the compiler driver’s command line too long for some operating systems.



This switch is deprecated.

-write-opts

The `-write-opts` (user options) switch directs the compiler to pass the user options (but not the input filenames) to the main driver via a temporary file which can help if the resulting main driver command line is too long.



This switch is deprecated.

Compiler Command-Line Interface

-xref *filename*

The `-xref` (cross-reference list) switch directs the compiler to write cross-reference listing information to the specified file. When more than one source file has been compiled, the listing contains information about the last file processed.

For each reference to a symbol in the source program, a line of the form

```
symbol-id name ref-code filename line-number column-number
```

is written to the named file.

The `symbol-id` identifier represents a unique decimal number for the symbol, and `ref-code` is a character from one the following (Table 1-12):

Table 1-12. ref-code Characters

Character	Meaning
D	Definition
d	Declaration
M	Modification
A	Address taken
U	Used
C	Changed (used and modified)
R	Any other type of reference
E	Error (unknown type of reference)



Please note that the compiler's `-xref` switch differs from the `-xref` switch used by the linker. Refer to the *VisualDSP++ 5.0 Linker and Utilities Manual* for more information.

C Mode (MISRA) Compiler Switch Descriptions

The following switches apply only to the C compiler. See [“MISRA-C Compiler” on page 1-91](#) for more information.

-misra

The `-misra` switch enables checking for MISRA-C Guidelines. Some rules or parts of rules are relaxed with this switch enabled. Rules relaxed by this option are 5.1, 5.7, 6.3, 6.4, 8.1, 8.2, 8.5, 10.3, 10.4, 10.5, 12.8, 13.7 and 19.7. This is explained in more detail, see [“Rules Descriptions” on page 1-95](#).

The `-misra` switch is not supported in conjunction with the `-w` and `-Werror|suppress|warn` switches. The switch predefines the `_MISRA_RULES` preprocessor macro.

-misra-linkdir *directory*

The `-misra-linkdir` switch specifies a directory in which to place `.misra` files. The default is a local directory called `MISRARepository`. The `.misra` files enables checking of violations of rules 5.5, 8.8 and 8.10.

-misra-no-cross-module

The switch implies `-misra`, but also disables checking for a number of rules that require the use of the prelinker to check across multiple modules for rule violation. The MISRA-C rules suppressed are 5.5, 8.8 and 8.10.

The `-misra-no-cross-module` switch is not supported in conjunction with the `-w` and `-Werror|remark|suppress|warn` switches.

-misra-no-runtime

The switch implies `-misra`, but also disables runtime-checking for MISRA-C rules 21, 17.1, 17.2 and 17.3. It limits the checking of rules 9.1, 12.8, 16.2 and 17.4.

Compiler Command-Line Interface

The `-misra-no-runtime` switch is not supported in conjunction with the `-w` and `-Werror|remark|suppress|warn` switches.

-misra-strict

The `-misra-strict` switch enables checking for MISRA-C Guidelines. The switch ensures a strict interpretation of the MISRA-C: 2004 Guidelines. See [“Rules Descriptions” on page 1-95](#) for more detail.

The `-misra-strict` switch is not supported in conjunction with the `-w` and `-Werror|remark|suppress|warn` switches. The switch predefines the `_MISRA_RULES` preprocessor macro.

-misra-suppress-advisory

The switch implies `-misra`, but suppresses the reporting of advisory rules.

The `-misra-suppress-advisory` switch is not supported in conjunction with the `-w` and `-Werror|remark|suppress|warn` switches.

-misra-testing

The switch implies `-misra` but also suppresses checking of MISRA-C rules 20.4, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12.

The `-misra-testing` switch is not supported in conjunction with the `-w` and `-Werror|remark|suppress|warn` switches.

-Wmis_suppress *rule_number* [, *rule_number*]

The `-Wmis_suppress` switch with a *rule_number* argument directs the compiler to suppress the specified diagnostic for a MISRA-C rule. The *rule_number* argument identifies the specific message to override.

-Wmis_warn *rule_number* [, *rule_number*]

The `-Wmis_warn` switch with a *rule_number* argument directs the compiler to override the severity of the specified diagnostic to produce a warning for a MISRA-C rule. The *rule_number* argument identifies the specific message to override.

C++ Mode Compiler Switch Descriptions

The following switches apply only to the C++ compiler.

-anach

The `-anach` (enable C++ anachronisms) directs the compiler to accept some language features that are prohibited by the C++ standard but still in common use. This is the default mode. Use the `-no-anach` switch ([on page 1-76](#)) for greater standard compliance.

The following anachronisms are accepted in the default C++ mode:

- Overload is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array may be specified in an array delete operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.

Compiler Command-Line Interface



- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as an un-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a `non-const` type may be initialized from a value of a different type. A temporary is created; it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a `non-const` class type may be initialized from an `rvalue` of the `class` type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following statements declare the overload of two functions named `f`.

```
int f(int);  
int f(x) char x; { return x; }
```

-check-init-order


It is not guaranteed that global objects requiring constructors are initialized before their first use in a program consisting of separately compiled units. The compiler will output warnings if these objects are external to the compilation unit and are used in dynamic initialization or in constructors of other objects. These warnings are not dependent on the `-check-init-order` switch.

In order to catch uses of these objects and to allow the opportunity for code to be rewritten, the `-check-init-order` (check initialization order) switch adds run-time checking to the code. This generates output to `stderr` that indicates uses of such objects are unsafe.

-  This switch generates extra code to aid development, and should not be used when building production systems.
-  Invoke this switch with the **Check initialization order** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.


-full-dependency-inclusion

The `-full-dependency-inclusion` switch ensures that when generating dependency information for implicitly-included `.cpp` files, the `.cpp` file will be re-included. This file is re-included only if the `.cpp` files are included more than once in the source (via re-inclusion of their corresponding header file). This switch is required only if your C++ sources files are compiled more than once with different macro guards.

-  Enabling this switch may increase the time required to generate dependencies.

-ignore-std

The `-ignore-std` switch directs the compiler to allow backwards compatibility to earlier versions of VisualDSP C++, which did not use namespace `std` to guard and encode C++ Standard Library names. By default, the header files and libraries now use namespace `std`.

-  Invoke this switch by clearing the **Use std:: namespace** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Source Language Settings** category.

-no-anach

The `-no-anach` (disable C++ anachronisms) switch directs the compiler to disallow some old C++ language features that are prohibited by the C++ standard. See the `-anach` switch ([on page 1-73](#)) for a full description of these features.

-no-implicit-inclusion

The `-no-implicit-inclusion` switch prevents implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

-no-rtti

The `-no-rtti` (disable run-time type identification) switch directs the compiler to disallow support for `dynamic_cast` and other features of ANSI/ISO C++ run-time type identification. This is the default mode. Use `-rtti` to enable this feature.

-no-std-templates

The `-no-std-templates` switch disables dependent name processing, i.e., the special lookup of names used in templates as required by the C++ standard.

-rtti

The `-rtti` (enable run-time type identification) switch directs the compiler to accept programs containing `dynamic_cast` expressions and other features of ANSI/ISO C++ run-time type identification. The switch also causes the compiler to define the macro `__RTTI` to 1. See also the `-no-rtti` switch.



Invoke this switch with the **C++ exceptions and RTTI** check box located in the VisualDSP++ **Project Options** dialog box, **Compile** page, **Language Settings** category.

-std-templates

The `-std-templates` switch enables dependent name processing, that is, the special lookup of names used in templates as required by the C++ standard.

Environment Variables Used by the Compiler

The compiler refers to a number of environment variables during its operation, as listed below. The majority of the environment variables identify *path names* to directories. You should be aware that placing network paths into these environment variables may adversely affect the time required to compile applications.

- **PATH**

This is your System search path, used to locate Windows applications when you run them. Windows uses this environment variable to locate the compiler when you execute it from the command line.

- **TMP**

This directory is used by the compiler for temporary files, when building applications. For example, if you compile a C file to an object file, the compiler first compiles the C file to an assembly file which can be assembled to create the object file. The compiler usually creates a temporary directory within the TMP directory into which to put such files. However, if the `-save-temps` switch is specified, the compiler creates temporary files in the current directory instead. This directory should exist and be writable. If this directory does not exist, the compiler issues a warning.

- **TEMP**

This environment variable is also used by the compiler when looking for temporary files, but only if TMP was examined and was not set or the directory that TMP specified did not exist.

Compiler Command-Line Interface

- **ADI_DSP**
The compiler locates other tools in the tool-chain through the VisualDSP++ installation directory, or through the `-path-install` switch. If neither is successful, the compiler looks in `ADI_DSP` for other tools.
- **CC21K_OPTIONS**
If this environment variable is set, and `CC21K_IGNORE_ENV` is not set, this environment variable is interpreted as a list of additional switches to be prepended to the command line. Multiple switches are separated by spaces or new lines. A vertical-bar (`|`) character may be used to indicate that any switches following it will be processed after all other command-line switches.
- **CC21K_IGNORE_ENV**
If this environment variable is set, `CC21K_OPTIONS` is ignored.

Data Type and Data Type Sizes

The sizes of intrinsic C/C++ data types are selected by Analog Devices so that normal C/C++ programs execute with hardware-native data types and therefore at high speed. [Table 1-13](#) shows the size used for each of the intrinsic C/C++ data types.

Table 1-13. Data Type Sizes for the ADSP-21xxx Processors

Type	Bit Size	Result of sizeof operator
int	32 bits signed	1
unsigned int	32 bits unsigned	1
long	32 bits signed	1
unsigned long	32 bits unsigned	1
char	32 bits signed	1
unsigned char	32 bits unsigned	1

Table 1-13. Data Type Sizes for the ADSP-21xxx Processors (Cont'd)

Type	Bit Size	Result of sizeof operator
short	32 bits signed	1
unsigned short	32 bits unsigned	1
pointer	32 bits	1
float	32 bits float	1
fract	32 bits fixed-point	1
double	either 32 or 64 bits float (default 32)	either 1 or 2 (default 1)
long double	64 bits float	2

Analog Devices does not support data sizes smaller than the addressable unit size on the processor. For the ADSP-21xxx processors, this means that both `short` and `char` have the same size as `int`. Although 32-bit `chars` are unusual, they do conform to the standard. For information about the `fract` data type, refer to [“C++ Fractional Type Support” on page 1-228](#).

Integer Data Types

On any platform, the basic type `int` is the native word size. For SHARC processors, it is 32 bits. Many library functions are available for 32-bit integers, and these functions provide support for the C/C++ data types `int` and `long int`. Pointers are the same size as `ints`. The `long long int` data type is not supported.

Floating-Point Data Types

For SHARC processors, the `float` data type is 32 bits long. The `double` data type is option-selectable for 32 or 64 bits. The C and C++ languages tend to default to `double` for constants and for many floating-point calculations. In general, double word data types run more slowly than 32-bit data types because they rely largely on software-emulated arithmetic.

Compiler Command-Line Interface

Type `double` poses a special problem. Without some special handling, many programs would inadvertently end up using slow-speed, emulated, 64-bit floating-point arithmetic, even when variables are declared consistently as `float`. In order to avoid this problem, Analog Devices provides the `-double-size[-32|-64]` switch ([on page 1-29](#)), which allows you to set the size of `double` to either 32 bits (default) or 64 bits. The 32-bit setting gives good performance and should be acceptable for most DSP programming. However, it does not conform fully to the ANSI C standard.

For a larger floating-point type, the `long double` data type provides 64-bit floating-point arithmetic.

For either size of `double`, the standard `#include` files automatically redefine the math library interfaces so that functions such as `sin` can be directly called with the proper size operands. Access to 64-bit floating-point arithmetic and libraries is always provided via `long double`.

Therefore,

```
float sinf (float);      /* 32-bit          */
double sin (double);     /* 32 or 64-bit */
```

For full descriptions of these functions and their implementation, see *VisualDSP++ 5.0 Run-Time Library Manual for SHARC Processors*.

Optimization Control

The general aim of compiler optimization is to generate correct code that executes quickly and is small in size. Not all optimizations are suitable for every application or possible all the time. Therefore, the compiler optimizer has a number of configurations, or optimization levels, which can be applied when needed. Each of these levels are enabled by one or more compiler switches (and VisualDSP++ project options) or pragmas.



Refer to Chapter 2, “[Achieving Optimal Performance from C/C++ Source Code](#)” for information on how to obtain maximal code performance from the compiler.

Optimization Levels

The following list identifies several optimization levels. The levels are notionally ordered with least optimization listed first and most optimization listed last. The descriptions for each level outline the optimizations performed by the compiler and identify any switches or pragmas required, or that have direct influence on the optimization levels performed.

- **Debug**

The compiler produces debug information to ensure that the object code matches the appropriate source code line. See “[-g](#)” on [page 1-35](#) and “[-Og](#)” on [page 1-50](#) for more information.

- **Default**

The compiler does not perform any optimization by default when none of the compiler optimization switches are used (or enabled in VisualDSP++ project options). Default optimization level can be enabled using the `optimize_off` pragma ([on page 1-171](#)).

- **Procedural optimizations**

The compiler performs advanced, aggressive optimization on each procedure in the file being compiled. The optimizations can be directed to favor optimizations for speed (`-O1` or `0`) or space (`-Os`) or a factor between speed and space (`-Ov`). If debugging is also requested, the optimization is given priority so the debugging functionality may be limited. See “[-O\[0|1\]](#)” on [page 1-49](#), “[-Os](#)” on [page 1-50](#), “[-Ov num](#)” on [page 1-50](#) and “[-Og](#)” on [page 1-50](#). Procedural optimizations for speed and space (`-O` and `-Os`) can be enabled in C/C++ source using the pragma `optimize_{for_speed|for_space}`. (For more information, see “[General Optimization Pragmas](#)” on [page 1-171](#).)

- **Profile-guided optimizations (PGO)**

The compiler performs advanced aggressive optimizations using profiler statistics (.pgo files) generated from running the application using representative training data. PGO can be used in conjunction with IPA and automatic inlining. See “-pguide” on [page 1-56](#) for more information.

Note that PGO is supported in the simulator **only**.

The most common scenario in collecting PGO data is to setup one or more simple File to Device streams where the File is a standard ASCII stream input file and the Device is any stream device supported by the simulator target, such as memory and peripherals. The PGO process can be broken down into the execution of one or more “Data Sets” where a Data Set is the association of zero or more input streams with a single .pgo output file. The user can create, edit and delete the Data Sets through the IDDE and then “run” the Data Sets with the click of one button to produce an optimized application. The PGO operation is handled via a new PGO submenu added to the top-level **Tools** menu:

Tools -> PGO -> Manage Data Sets.

[For more information, see “Using Profile-Guided Optimization” in Chapter 2, Achieving Optimal Performance from C/C++ Source Code.](#)



Note the requirement for allowing command-line arguments in your project when using PGO. For further details refer to “[Support for argv/argc](#)” on [page 1-282](#).

- **Automatic Inlining**

The compiler automatically inlines C/C++ functions which are not necessarily declared as inline in the source code. It does this when it has determined that doing so reduces execution time. How aggressively the compiler performs automatic inlining is controlled using the -Ov switch. Automatic inlining is enabled using the -Oa

switch which additionally enables procedural optimizations (-O). See “-Oa” on page 1-49, “-Ov num” on page 1-50, “-O[0|1]” on page 1-49 and “Function Inlining” on page 1-108 for more information.



When remarks are enabled, the compiler produces a remark to indicate each function that is inlined.

- **Interprocedural Optimizations**

The compiler performs advanced, aggressive optimization over the whole program, in addition to the per-file optimizations in procedural optimization. The *interprocedural analysis* (IPA) is enabled using the -ipa switch which additionally enables procedural optimizations (-O). See “Interprocedural Analysis”, “-ipa” on page 1-39 and “-O[0|1]” on page 1-49 for more information.

The compiler optimizer attempts to vectorize loops when it is safe to do so. When IPA is used it can identify additional safe candidates for vectorization which might not be classified as safe at a procedural optimization level. Additionally, there may be other loops that are known to be safe candidates for vectorization which can be identified to the compiler with use of various pragmas. (See “Loop Optimization Pragmas” on page 1-166.)

Using the various compiler optimization levels is an excellent way of improving application performance. However consideration should be given to how applications are written so that compiler optimizations are given the best opportunity to be productive. These issues are the topic of Chapter 2, “Achieving Optimal Performance from C/C++ Source Code”.

Interprocedural Analysis

The `cc21k` compiler has a capability called *interprocedural analysis* (IPA), an optimization that allows the compiler to optimize across translation units instead of within just one translation unit. This capability effectively allows the compiler to see all of the source files that are used in a final link at compilation time and make use of that information when optimizing.

Interprocedural analysis is enabled by selecting the **Interprocedural Analysis** check box in the VisualDSP++ **Project Options** dialog box, **Compile** tab, **General** category, or by specifying the `-ipa` command-line switch.

The `-ipa` switch automatically enables the `-O` switch to turn on optimization. (See “[-ipa](#)” on page 1-39.)

Use of the `-ipa` switch generates additional files along with the object file produced by the compiler. These files have `.ipa` filename extensions and should not be deleted manually unless the associated object file is also deleted.

All of the `-ipa` optimizations are invoked after the initial link, whereupon a special program called the prelinker reinvokes the compiler to perform the new optimizations, recompiling source files where necessary, to make use of gathered information.



Because a file may be recompiled by the prelinker, do not use the `-S` option to see the final optimized assembler file when `-ipa` is enabled. Instead, use the `-save-temps` switch ([on page 1-60](#)), so that the full compile/link cycle can be performed first.

Interaction with Libraries

When IPA is enabled, the compiler examines all of the source files to build up usage information about all of the function and data items. It then uses that information to make additional optimizations across all of the source files by recompiling where necessary.

Because IPA operates only during the final link, the `-ipa` switch has no benefit when initially compiling source files to object format for inclusion in a library. IPA gathers information about each file and embeds this within the object format, but cannot make use of it at this point, because the library contents have not yet been used in a specific context.

When IPA is invoked during linking, it will recover the gathered information from all linked-in object files that were built with `-ipa`, and where necessary and possible, will recompile source files to apply additional optimizations. Modules linked in from a library are not recompiled in this manner, as source is not available for them. Therefore, the gathered information in a library module can be used to further optimize application sources, but does not provide a benefit to the library module itself.

If a library module makes references to a function in a user module in the program, this will be detected during the initial linking phase, and IPA will not eliminate the function. If the library module was not compiled with `-ipa`, IPA will not make any assumptions about how the function may be called, so the function may not be optimized as effectively as if all references to it were in source code visible to IPA, or from library modules compiled with `-ipa`.

Controlling Silicon Revision and Anomaly Workarounds within the Compiler

The compiler provides three switches which specify that code produced by the compiler will be generated for a specific revision of a specific processor, and appropriate silicon revision targeted system run time libraries will be linked against. Targeting a specific processor allows the compiler to produce code that avoids specific hardware errata reported against that revision. For the simplest control, use the `-si-revision` switch which automatically controls compiler workarounds.



The compiler cannot apply errata workarounds to code inside `asm()` constructs.

Compiler Command-Line Interface

When developing using the IDDE, the silicon revision within a project is set to a default value of `Automatic`. Using a silicon revision of `Automatic` will select a value for the `-si-revision` switch based on the hardware connected and the session type that is currently in use. This will enable all errata workarounds for the determined silicon revision.

This section describes:

- [“Using the -si-revision Switch” on page 1-86](#)
- [“Using the -workaround Switch” on page 1-88](#)
- [“Using the -no-workaround Switch” on page 1-89](#)
- [“Interactions Between the Silicon Revision and Workaround Switches” on page 1-89](#)

Using the -si-revision Switch

The `-si-revision version` (silicon revision) switch directs the compiler to build for a specific hardware revision. Any errata workarounds available for the targeted silicon revision will be enabled. The parameter `version` represents a silicon revision of the processor specified by the `-proc` switch ([on page 1-57](#)).

For example,

```
cc21k -proc ADSP-21161 -si-revision 0.1 prog.c
```

If silicon version `none` is used, then no errata workarounds are enabled, whereas specifying silicon version `any` will enable all errata workarounds for the target processor.


If the `-si-revision` switch is not used, the compiler will build for the latest known silicon revision for the target processor and any errata workarounds which are appropriate for the latest silicon revision will be enabled.


Run-time libraries built without any errata workarounds are located in the platform's `lib` sub-directory; for example, `212xx/lib`. Within the `lib` sub-directory, there are library directories for each silicon revision; these libraries have been built with errata workarounds appropriate for the silicon revision enabled. Note that an individual set of libraries may cover more than one specific silicon revision, so if several silicon revisions are affected by the same errata, then one common set of libraries might be used.

The `__SILICON_REVISION__` macro is set by the compiler to two hexadecimal digits representing the major and minor numbers in the silicon revision. For example, 1.0 becomes `0x100` and 10.21 becomes `0xa15`.

If the silicon revision is set to any, the `__SILICON_REVISION__` macro is set to `0xffff` and if the `-si-revision` switch is set to `none` the compiler will not set the `__SILICON_REVISION__` macro.

The compiler driver will pass the `-si-revision` switch, as specified in the command line, when invoking other tools in the VisualDSP++ toolchain.

 On ADSP-2116x processors, the workaround for the shadow write FIFO anomaly is only required under certain circumstances and is therefore not enabled by default. If this workaround is required, the `-workaround swfa` switch should be used. On the ADSP-21161 processor, anomaly #45 can only occur if the code is executed from external memory, and accordingly the workaround for this anomaly is not enabled by default. If this workaround is required, the `-workaround 21161-anomaly-45` switch should be used.

 Visit <http://www.analog.com/processors/technicalSupport/ICAnomalies.html> to get more information on specific anomalies (including anomaly IDs).

Using the -workaround Switch

The `-workaround` switch enables code generator workarounds for specific hardware defects. [Table 1-14 on page 1-88](#) lists valid workarounds.

Table 1-14. Valid Workarounds

Workaround	Description
21161-anomaly-45	Specifies that the compiler does not generate conditional DAG1 instructions
2126x-anomaly-4	Specifies that the compiler will avoid producing anomalous counter based single instruction loops with PMDA access. The code is either rescheduled or a NOP is inserted. The compiler also defines the macro <code>__WORKAROUND__2126X_ANOMALY4__</code> at the source, assembly and link build stages when this workaround is enabled.
2136x-multi	Specifies that the compiler does not generate code that will produce the badly executed stalls associated with this anomaly. The compiler also defines the macro <code>__WORKAROUND__2136X_ANOMALY2__</code> at the source, assembly and link build stages when this workaround is enabled.
2136x-mem-write	Instructs the compiler to reschedule code or insert NOPs to avoid memory write operations that may fail due to this anomaly. The compiler also defines the macro <code>__WORKAROUND__2136X_ANOMALY3__</code> at the source, assembly and link build stages when this workaround is enabled.
all	Indicates that the compiler enables all known workarounds. In addition to the compiler generating errata safe code, this directs the default <code>.ldf</code> files to link against run-time libraries that are safe for all workarounds.
dag-stall	Specifies that the compiler attempts to avoid stalls produced by the assignment and immediate use of a DAG register by scheduling other instructions (or NOPs) in between. This workaround is for the ADSP-2106x and ADSP-2116x chips only.

Table 1-14. Valid Workarounds (Cont'd)

Workaround	Description
rframe	Specifies that the compiler shall not generate the <code>rframe</code> instruction as part of the function return sequence. This workaround is for the ADSP-21160 chip only.
swfa	Specifies that the compiler does not generate code that could be affected by the shadow write FIFO anomaly on ADSP-2116x chips

Using the -no-workaround Switch

The `-no-workaround workaroundID[,workaroundID ...]` switch disables compiler code generator workarounds for specific hardware errata. Current valid workarounds are listed in [Table 1-14 on page 1-88](#) (Workarounds may change).

The `-no-workaround` switch can be used to disable workarounds enabled via the `-si-revision version` or `-workaround workaroundID` switch.

All workarounds can be disabled by providing `-no-workaround` with all valid workarounds for the selected silicon revision or by using the option `-no-workaround all`. Disabling all workarounds via the `-no-workaround` switch will link against libraries with no silicon revision in cases where the silicon revision is not `none`.

Interactions Between the Silicon Revision and Workaround Switches

The interactions between `-si-revision`, `-workaround` and `-no-workaround` can only be determined once all the command line arguments have been parsed.

Compiler Command-Line Interface

To this effect options will be evaluated as follows:

1. The `-si-revision version` is parsed to determine which revision of the run-time libraries the application will link against. It also produces an initial list of all the default compiler errata workarounds to enable.
2. Any additional workarounds specified with the `-workaround` switch will be added to the errata list.
3. Any workarounds specified with `-no-workaround` will then be removed from this list.
4. If silicon revision is not `none` or if any workarounds were declared via `-workaround`, the macro `__WORKAROUNDS_ENABLED` will be defined at compile and assembly and link stages, even if `-no-workaround` disables all workarounds.

MISRA-C Compiler

This section provides an overview of MISRA-C compiler and MISRA-C 2004 Guidelines.

MISRA-C Compiler Overview

The Motor Industry Software Reliability Association (MISRA) in 1998 published a set of guidelines for the C programming language to promote best practice in developing safety related electronic systems in road vehicles and other embedded systems. The latest release of MISRA-C:2004 has addressed many issues raised in the original guidelines specified in MISRA-C:1998. Complex rules are now split into component parts. There are 121 mandatory and 20 advisory rules. The compiler issues a discretionary error for mandatory rules and a warning for advisory rules. More information on MISRA-C can be obtained at <http://www.misra.org.uk/>.

The compiler detects violations of the MISRA-C rules at compile-time, link-time and runtime. It has full support for the MISRA-C 2004 Guidelines. The majority of MISRA-C rules are easy to interpret. Those that require further explanation can be found in [“Rules Descriptions” on page 1-95](#).

As a documented extension, the compiler supports the type qualifiers `__pm` and `__dm` (see [“Dual Memory Support Keywords \(pm dm\)” on page 1-130](#)). No other language extensions are supported when MISRA checking is enabled. Common extensions, such as the keywords `section` and `inline`, are not allowed in the MISRA-C mode, but the same effects can be achieved by using pragmas [“#pragma section/#pragma default_section” on page 1-201](#) and [“#pragma inline” on page 1-194](#). Rules can be suppressed by the use of command-line switches or the MISRA-C extensions to [“Diagnostic Control Pragmas” \(see on page 1-207\)](#).



The run-time checking that is used for validating a number of rules should not be used in production code. The cost of detecting these violations is expensive in both run-time performance and code size.

Refer to [Table 1-4 on page 1-20](#) for the list of MISRA-C command-line switches.

MISRA-C Compliance

The MISRA-C:2004 Guidelines Forum (visit <http://www.misra.org.uk/>) is an essential reference for ensuring that code developed or requiring modification complies to these guidelines. A rigorous checking tool such as this compiler makes achieving compliance a lot easier than using a less capable tool or simply relying on manual reviews of the code. The MISRA-C:2004 Guidelines Forum describes a compliance matrix that a developer uses to ensure that each rule has a method of detecting the rule violation. A compliance checking tool is a vital component in detecting rule violations. It is recognized in the guidelines document that in some circumstances it may be necessary to deviate from the given rules. A formal procedure has to be used to authorize these deviations rather than an individual programmer having to deviate at will.

Using the Compiler to Achieve Compliance

The VisualDSP++ compiler is one of the most comprehensive MISRA-C: 2004 compliance checking tools available. The compiler has various command-line switches and “[Diagnostic Control Pragmas](#)” (see [on page 1-207](#)) to enable you to achieve MISRA-C: 2004 compliance.

During development, it is recommended that the application is built with maximum compliance enabled.

Use the `-misra-strict` command-line switch to detect the maximum number of rule violations at compile-time. However, if existing code is being modified, using `-misra-strict` may result in a lot of errors and warnings. The majority are usually common rule violations that are

mainly advisory and typically found in header files as a result of macro expansion. These can be suppressed using the `-misra` command-line switch. This has the potential benefit of focussing change on individual source file violations, before changing headers that may be shared by more than one project.

The `-misra-no-cross-module` command-line switch disables checking rule violations that occur across source modules. During development, some external variables may not be fully utilized and rather than add in artificial uses to avoid rule violations, use this switch.

The `-misra-no-runtime` command-line switch disables the additional run-time overheads imposed by some rules. During development these checks are essential in ensuring code executes as expected. Use this switch in release mode to disable the run-time overheads.

You can use the `-misra-testing` command-line switch during development to record the behavior of executable code. Although the MISRA-C: 2004 Guidelines do not allow library functions, such as those that are defined in header `<stdio.h>`, it is recognized that they are an essential part of validating the development process.

During development, it is likely that you will encounter areas where some rule violations are unavoidable. In such circumstances you should follow the procedure regarding rule deviations described in the MISRA-C: 2004 Guidelines Forum. Use the `-Wmis_suppress` and `-Wmis_warn` switches to control the detection of rule violations for whole source files. Finer control is provided by the diagnostic control pragmas. These pragmas allow you to suppress the detection of specified rule violations for any number of C statements and declarations.

Example

```
#include <misra_types.h>
#include <def21061.h>
#include "proto.h" /* prototype for func_state and my_state */
int32_t func_state(int32_t state)
```

```
{
return state & TIMOD1;
/* both operands signed, violates rule 12.7 */
}
```

```
#define my_flag 1
```

```
int32_t my_state(int32_t state)
{
return state & my_flag;
/* both operands signed, violates rule 12.7 */
}
```

In the above example, <def21061.h> uses signed masks and signed literal values for register values. The code is meaningful and trusted in this context. You may suppress this rule and document the deviation in the code. For code violating the rule that is not from the system header, you may wish to rewrite the code:

```
#include <misra_types.h>
#include <def21061.h>
#include "proto.h" /* prototype for func_state and my_state */

#ifdef _MISRA_RULES
#pragma diag(push)
#pragma diag(suppress:misra_rule_12_7:"Using the def file is
a safe and justified deviation for rule 12.7")

#endif /* _MISRA_RULES */

int32_t func_state(int32_t state)
{
return state & TIMOD11;
/* both operands signed, violates rule 12.7 */
}

#ifdef _MISRA_RULES
#pragma diag(pop)
/* allow violations of 12.7 to be detected again */
#endif /* _MISRA_RULES */
```

```
#define my_flag 1u

uint32_t my_state(uint32_t state)
{
    return state & my_flag; /* o.k both unsigned */
}
```

Rules Descriptions

The following are brief explanations of how some of the MISRA-C rules are supported and interpreted in this VisualDSP++ release due to the fact that some rules are handled in a nonstandard way, or some are not handled at all:



Since the data types `char`, `short` and `int` are all represented as 32-bit integers on the SHARC architecture, MISRA rules relating to the size of variables may not be issued.

- **Rule 1.4 (required): The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.**

The compiler and linker fully support this requirement.

Rule 1.5 (required): Floating-point implementations should comply with a defined floating-point standard.

Refer to [“Floating-Point Data Types” on page 1-79](#).

Rule 2.4 (advisory): Sections of code should not be “commented out”.

A diagnostic is reported if one of the following is encountered inside of a comment.

- character ‘{’ or ‘}’
- character ‘;’ followed by a new-line character

Rule 5.1 (required): Identifiers (internal and external) shall not rely on the significance of more than 31 characters.

This rule is only enforced when the `-misra-strict` compiler switch is enabled ([on page 1-72](#)).

Rule 5.5 (advisory): No object or function identifier with static storage duration should be reused.

This rule is enforced by the compiler prelinker. The compiler generates `.misra` extension files that the prelinker uses to ensure that the same identifier is not used at file-scope within another module. This rule is not enforced if the `-misra-no-cross-module` compiler switch is specified ([on page 1-71](#)).

Rule 5.7 (advisory): No identifier shall be reused.

This rule is limited to a single source file. The rule is only enforced when the `-misra-strict` compiler switch is enabled ([on page 1-72](#)).

Rule 6.3 (advisory): typedefs that indicate size and signedness should be used in place of basic types.

The typedefs for the basic types are provided by the system header file `<misra_types.h>`. The rule is only enforced when the `-misra-strict` compiler switch is enabled ([on page 1-72](#)).

Rule 6.4 (advisory): Bit fields shall only be defined to be of type unsigned int or signed int.

The rule regarding the use of plain `int` is only enforced when the `-misra-strict` compiler switch is enabled ([on page 1-72](#)).

Rule 8.1 (required): Functions shall have prototype declarations and the prototype shall be visible at both the function definition and the call.

For static and inline functions this rule is only enforced when the `-misra-strict` compiler switch is enabled ([on page 1-72](#)).

- **Rule 8.5 (required): There shall be no definitions of objects or functions in a header file.**

This rule does not apply to inline functions.

Rule 8.8 (required): An external object or function shall be declared in one and only one file.

This rule is enforced by the compiler prelinker. The compiler generates `.misra` extension files that the prelinker uses to ensure that the global is used in another file. The rule is not enforced if the `-misra-no-cross-module` switch is enabled ([on page 1-71](#)).

Rule 8.10 (required): All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.

This rule is enforced by the compiler prelinker. The compiler generates `.misra` extension files that the prelinker uses to ensure that the global is used in another file. The rule is not enforced if the `-misra-no-cross-module` switch is enabled ([on page 1-71](#)).

Rule 9.1 (required): All automatic variables shall have been assigned a value before being used.

The compiler attempts to detect some instances of violations of this rule at compile-time. There is additional code added at run-time to detect unassigned scalar variables. The additional Integral types with a size less than an `int` are not checked by the additional run-time code. The run-time code is not added if the `-misra-no-runtime` compiler switch is enabled ([on page 1-71](#)).

Rule 10.5 (required): If the bitwise operators `~` and `<<` are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.

When constant-expressions violate this rule, they are only detected when the `-misra-strict` compiler switch is enabled ([on page 1-72](#)).

Rule 11.3 (advisory): A cast shall not be performed between a pointer type and an integral type.

The compiler always allows a constant of integral type to be cast to a pointer to a volatile type.

```
volatile int32_t *n;
```

```
n = (volatile int32_t *)10;
```

There is only one case where this rule is not applied.

```
int32_t *n;  
n = (int32_t *)10;
```

- **Rule 12.4 (required): The right-hand operand of a logical && or || operator shall not contain side-effects.**

A function call used as the right-hand operand will not be faulted if it is declared with an associated `#pragma pure` directive.

- **Rule 12.7 (required): Bitwise operators shall not be applied to operands whose underlying type is signed.**

The compiler will not enforce this rule if the two operands are constants.

Rule 12.8 (required): The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.

If the right-hand operand is not a constant expression, the violation will be checked by additional run-time code when `-misra-no-runtime` is not enabled. If both operands are constants, the rule is only enforced when the `-misra-strict` compiler switch is enabled ([on page 1-72](#)).

- **Rule 12.12 (required): The underlying bit representations of floating-point values shall not be used.**

MISRA-C rules such as 11.4 prevent casting of bit-patterns to floating-point values. Hexadecimal floating-point constants are also not allowed when MISRA-C switches are enabled.

- **Rule 13.2 (advisory): Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.**

The compiler treats variables which use the type `bool` (a typedef is declared in `<misra_types.h>`) as “Effectively Boolean” and will not raise an error when these are implicitly tested as zero, as follows:

```
bool b = 1;
if(bool)
    ...;
```

Rule 13.7 (required): Boolean operations whose results are invariant shall not be used.

The compiler does not detect cases where there is a reliance on more than one conditional statement. Constant expressions violating the rule are only detected when the `-misra-strict` compiler switch is enabled ([on page 1-72](#)).

Rule 16.2 (required): Functions shall not call themselves, either directly or indirectly.

A compile-time check is performed for a single file. Run-time code is added to ensure that functions do not call themselves directly or indirectly, but this code is not generated if the `-misra-no-runtime` compiler switch is enabled ([on page 1-71](#)).

Rule 16.4 (required): The identifiers used in the declaration and definition of a function shall be identical.

A declaration of a parameter name may have one leading underscore that the definition does not contain. This is to prevent name clashing. If the `-misra-strict` compiler switch is enabled ([on page 1-72](#)), the underscore is significant and results in the violation of this rule.

Rule 16.5 (required): Functions with no parameters shall be declared and defined with parameter type void.

Function `main` shall only be reported as violating this rule if the `-misra-strict` compiler switch is enabled ([on page 1-72](#)).

- **Rule 16.10 (required): If a function returns error information, then the error information shall be tested.**

A function declared with return type `bool`, which is a typedef declared in header file `<misra_types.h>` will be faulted if the result of the call is not used.

- **Rule 17.1 (required): Pointer arithmetic shall only be applied to pointers that address an array or array element.**
Checking is performed at run-time. A run-time function looks at the value of the pointer and checks to see whether it violates this rule.
- **Rule 17.2 (required): Pointer subtraction shall only be applied to pointers that address elements of the same array.**
Checking is performed at runtime. A run-time function looks at the value of the pointers and checks to see whether it violates this rule.
- **Rule 17.3 (required): >, >=, <, <= shall not be applied to pointers that address elements of different arrays.**
Checking is performed at run-time. A run-time function looks at the value of the pointers and checks to see whether it violates this rule.
- **Rule 17.4 (required): Array indexing shall be the only allowed form of pointer arithmetic.**
Checking is performed at runtime to ensure the object being indexed is an array. A run-time function looks at the value of the pointers and checks to see whether it violates this rule.

All other forms of pointer arithmetic are reported at compile-time as violations of this rule.

- **Rule 17.6 (required): The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.**
Rule is not enforced under the following circumstances: if the address of a local variable is passed as a parameter to another function, the compiler cannot detect whether that address has been assigned to a global object.

- **Rule 18.2 (required):** An object shall not be assigned to an overlapping object.

The rule is not enforced by the compiler.

- **Rule 18.3 (required):** An area of memory shall not be reused for unrelated purposes.

The rule is not enforced by the compiler.

Rule 19.4 (required): C macros shall only expand to a braced initializer, a constant, a string literal, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

Use of `#pragma diag(suppress:misra_rule_19_4)` will suppress violations of this rule for any macro expansion during the scope of the suppression. If a macro is defined within the scope of the suppression, then the macro expansion will not be detected for violation of rule 19.4 even if the expansion point does not suppress the rule. [“Diagnostic Control Pragmas” on page 1-207](#)

- **Rule 19.7 (advisory):** A function shall be used in preference to a function-like macro.

The rule is only enforced when the compiler option `-misra-strict` is enabled ([on page 1-72](#)).

- **Rule 19.15 (required):** Precautions shall be taken in order to prevent the contents of a header file being included twice.

The compiler will report this violation if a header file is included more than once and does not prevent redeclarations of types, variables or functions.

- **Rule 20.3 (required):** The validity of values passed to library functions shall be checked.

This is not enforced by the compiler. The rule puts the responsibility on the programmer.

- **Rule 20.4 (required): Dynamic heap memory allocation shall not be used.**

Prototype declarations for functions performing heap allocation should be declared with an associated `#pragma misra_func(heap)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- **Rule 20.7 (required): The `setjmp` macro and `longjmp` function shall not be used.**

Prototype declarations for these should be declared with an associated `#pragma misra_func(jmp)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- **Rule 20.8 (required): The signal handling facilities of `<signal.h>` shall not be used.**

Prototype declarations for functions in this header should be declared with an associated `#pragma misra_func(handler)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- **Rule 20.9 (required): The input/output library `<stdio.h>` shall not be used.**

Prototype declarations for functions in this header should be declared with an associated `#pragma misra_func(io)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- **Rule 20.10 (required): The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.**

Prototype declarations for these functions should be declared with an associated `#pragma misra_func(string_conv)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

- **Rule 20.11 (required):** The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.
 Prototype declarations for these functions should be declared with an associated `#pragma misra_func(system)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.
- **Rule 20.12 (required):** The time handling functions of library `<time.h>` shall not be used.
 Prototype declarations for these functions should be declared with an associated `#pragma misra_func(time)` directive. This directive allows the compiler to detect violations of this rule when these functions are used.

Rule 21.1 (required): Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.

The compiler performs some static checks on uses of unassigned variables before conditional code and use of constant expressions. The compiler performs run-time checks for arithmetic errors, such as division by zero, array bound errors, unassigned variable checking and pointer dereferencing. Run-time checking has a negative effect on code performance. The `-misra-no-runtime` compiler switch turns off the run-time checking ([on page 1-71](#)).

C/C++ Compiler Language Extensions

The compiler supports a set of extensions to the ANSI standard for the C and C++ languages. These extensions add support for DSP hardware and allow some C++ programming features when compiling in C mode. Most extensions are also available when compiling in C++ mode.

This section contains:

- [“Function Inlining” on page 1-108](#)
- [“Inline Assembly Language Support Keyword \(asm\)” on page 1-113](#)
- [“Dual Memory Support Keywords \(pm dm\)” on page 1-130](#)
- [“Bank Type Qualifiers” on page 1-135](#)
- [“Placement Support Keyword \(section\)” on page 1-136](#)
- [“Placement of Compiler-Generated Code and Data” on page 1-137](#)
- [“Boolean Type Support Keywords \(bool, true, false\)” on page 1-139](#)
- [“Pointer Class Support Keyword \(restrict\)” on page 1-139](#)
- [“Variable-Length Array Support” on page 1-140](#)
- [“Long Identifiers” on page 1-141](#)
- [“Non-Constant Initializer Support” on page 1-142](#)
- [“Indexed Initializer Support” on page 1-142](#)
- [“Aggregate Constructor Expression Support” on page 1-144](#)
- [“Preprocessor Generated Warnings” on page 1-145](#)
- [“C++ Style Comments” on page 1-145](#)

- “Compiler Built-In Functions” on page 1-146
- “Pragmas” on page 1-156
- “GCC Compatibility Extensions” on page 1-219
- “C++ Fractional Type Support” on page 1-228
- “Saturated Arithmetic” on page 1-230
- “SIMD Support” on page 1-231
- “Accessing External Memory on ADSP-2126X and 2136X Processors” on page 1-246
- “Support for Interrupts” on page 1-247
- “Migrating .ldf Files From Previous VisualDSP++ Installations” on page 1-254

The additional keywords that are part of the C/C++ extensions do not conflict with any ANSI C/C++ keywords. The formal definitions of these extension keywords are prefixed with a leading double underscore (`__`). Unless the `-no-extra-keywords` command-line switch (on page 1-45) is used, the compiler defines the shorter form of the keyword extension that omits the leading underscores.

This section describes only the shorter forms of the keyword extensions, but in most cases you can use either form in your code. For example, all references to the `inline` keyword in this text appear without the leading double underscores, but you can interchange `inline` and `__inline` in your code.

You might need to use the longer form (such as `__inline`) exclusively if porting a program that uses the extra Analog Devices keywords as identifiers. For example, a program might declare local variables, such as `pm` or `dm`. In this case, use the `-no-extra-keywords` switch, and if you need to declare a function as `inline`, or allocate variables to memory spaces, you can use `__inline` or `__pm/__dm` respectively.

This section provides an overview of the extensions, with brief descriptions, and directs you to text with more information on each extension.

Table 1-15 provides a brief description of each keyword extension and directs you to sections of this chapter that document the extensions in more detail. Table 1-16 provides a brief description of each operational extension and directs you to sections that document these extensions in more detail.

Table 1-15. Keyword Extensions

Keyword extensions	Description
<code>inline</code>	Directs the compiler to integrate the function code into the code of the calling function(s). For more information, see “Function Inlining” on page 1-108.
<code>asm()</code>	Places ADSP-21xxx assembly language instructions directly in your C/C++ program. For more information, see “Inline Assembly Language Support Keyword (asm)” on page 1-113.
<code>dm</code>	Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to Data Memory (DM). For more information, see “Dual Memory Support Keywords (pm dm)” on page 1-130.
<code>pm</code>	Specifies the location of a static or global variable or qualifies a pointer declaration “*” as referring to Program Memory (PM). For more information, see “Dual Memory Support Keywords (pm dm)” on page 1-130.
<code>section("string")</code>	Specifies the section in which an object or function is placed. The <code>section</code> keyword has replaced the <code>segment</code> keyword of the previous releases of the compiler software. For more information, see “Placement Support Keyword (section)” on page 1-136.
<code>bool, true, false</code>	A boolean type. For more information, see “Boolean Type Support Keywords (bool, true, false)” on page 1-139.
<code>restrict</code> keyword	Specifies restricted pointer features. For more information, see “Pointer Class Support Keyword (restrict)” on page 1-139.

Table 1-16. Operational Extensions

Operation extensions	Description
Variable-length arrays	Support for variable-length arrays lets you use arrays whose length is not known until run time. For more information, see “Variable-Length Array Support” on page 1-140.
Long identifiers	Supports identifiers of up to 1022 characters in length. For more information, see “Long Identifiers” on page 1-141.
Non-constant initializers	Support for non-constant initializers lets you use non-constants as elements of aggregate initializers for automatic variables. For more information, see “Non-Constant Initializer Support” on page 1-142.
Indexed initializers	Support for indexed initializers lets you specify elements of an aggregate initializer in arbitrary order. For more information, see “Indexed Initializer Support” on page 1-142.
Aggregate constructor expressions	Support for aggregate assignments lets you create an aggregate array or structure value from component values within an expression. For more information, see “Aggregate Constructor Expression Support” on page 1-144.
<code>fract</code> data type (C++ mode)	Support for the fractional data type, fractional and saturated arithmetic. For more information, see “C++ Fractional Type Support” on page 1-228.
Preprocessor-generated warnings	Lets you generate warning messages from the preprocessor. For more information, see “Preprocessor Generated Warnings” on page 1-145.
C++ style comments	Allows for “//” C++ style comments in C programs. For more information, see “C++ Style Comments” on page 1-145.

Function Inlining

The `inline` keyword directs the compiler to integrate the code for the function you declare as `inline` into the code of its callers. Inline function support and the `inline` keyword is a standard feature of C++; the compiler provides this keyword as a C extension.

This keyword eliminates the function call overhead and increases the speed of your program's execution. Argument values that are constant and that have known values may permit simplifications at compile time so that not all of the inline function's code needs to be included.

The following example shows a function definition that uses the `inline` keyword.

```
inline int max3 (int a, int b, int c) {  
    return max (a, max(b, c));  
}
```

The compiler can decide not to inline a particular function declared with the `inline` keyword, with a diagnostic remark `cc1462` issued if the compiler chooses to do this. The diagnostic can be raised to a warning by use of the `-Wwarn` switch. [For more information, see “-W{error|remark|suppress|warn} number\[,number ...\]” on page 1-67.](#)

Function inlining can also occur by use of the `-Oa` (automatic function inlining) switch ([For more information, see “-Oa” on page 1-49.](#)), which enables the inline expansion of C/C++ functions that are not necessarily declared inline in the source code. The amount of auto-inlining the compiler performs is controlled using the `-Ov` (optimize for speed versus size) switch.

The compiler follows a specific order of precedence when determining whether a call can be inlined. The order is:

1. If the definition of the function is not available (for example, it is a call to an external function), the compiler cannot inline the call.
2. If the `-never-inline` switch has been specified (on page 1-42), the compiler will not inline the call. If the call is to a function that has `#pragma always_inline` specified (see “Inline Control Pragas” on page 1-192), a warning will also be issued.
3. If the call is to a function that has `#pragma never_inline` specified, the call will not be inlined.
4. If the call is via a pointer-to-function, the call will not be inlined unless the compiler can prove that the pointer will always point to the same function definition.
5. If the call is to a function that has a variable number of arguments, the call will not be inlined.
6. If the module contains `asm` statements at global scope (outside function definitions), the call may not be inlined because the `asm` statement restricts the compiler’s ability to reorder the resulting assembly output.
7. If the call is to a function that has `#pragma always_inline` specified, the call is inlined. If the call exceeds the current speed/space ratio limits, the compiler will issue a warning, but will still inline the call.
8. If the call is to a function that has the `inline` qualifier, or has `#pragma inline` specified, and the `-always-inline` switch (on page 1-26) has been specified, the compiler will inline the call. If the call exceeds the current speed/space ratio limits, the compiler will issue a warning, but will still inline the call.
9. If the call is to a function that has the `inline` qualifier or has `#pragma inline` specified and optimization is enabled, the called function will be compared against the current speed/size ratio lim-

its for code size and stack size. The calling function will also be examined against these limits. Depending on the limits and the relative sizes of the caller and callee, the inlining may be rejected.

10. If the call is to a function that does not have the `inline` qualifier or `#pragma inline`, and does not have `#pragma weak_entry`, then if the `-Oa` switch has been specified to enable automatic inlining, the called function will be considered as a possible candidate for inlining, according to the current speed/size ratio limits, as if the `inline` qualifier were present.

The compiler bases its code-related speed/size comparisons on the `-Ov` switch. When `-Ov` is in the range 1...100, the compiler performs a calculation upon the size of the generated code using the `-Ov` value, and this will determine whether the generated code is “too large” for inlining to occur. When `-Ov` has the value 1, only very small functions are considered small enough to inline; when `-Ov` has the value 100, larger functions are more likely to be considered suitable as well.

When `-Ov` has the value 0, the compiler is optimizing for space. The speed/space calculation will only accept a call for inlining if it appears that the inlining is likely to result in less code than the call itself would (although this is an approximation, since the inlining process is a high-level optimization process, before actual machine instructions have been selected).

The inlining process also considers the required stack size while inlining. A function that has a local array of 20 integers needs such an array for each inlined invocation, and if inlined many times, the cumulative effect on overall stack requirements can be significant. Consequently, the compiler considers both the stack space required by the called function, and the total stack space required by the caller; either may reach a limit at which the compiler determines that inlining the call would not be beneficial. The stack size analysis is not subject to the `-Ov` switch.

Inlining and Optimization

The inlining process operates regardless of whether optimization has been selected (although if optimization is not enabled, then inlining will only happen when forced by `#pragma always_inline` or the `-always-inline` switch). The speed/size calculation still has an effect, although an optimized function is likely to have a different size from a non-optimized one, which is smaller (and therefore more likely to be inlined) and is dependent on the kind of optimization done.

A non-optimized function has loads and stores to temporary values which are optimized away in the optimized version, but an optimized function may have unrolled or vectorized loops with multiple variants, selected at run-time for the most efficient loop kernel, so an optimized function may run faster, but not be smaller.

Given that the optimization emphasis may be changed within a module – or even turned off completely – by the optimization pragmas, it is possible for either, both, or neither of the caller and callee to be optimized. The inlining process still operates, and is only affected by this in as far as the speed/size ratios of the resulting functions are concerned.

Inlining and Out-of-Line Copies

If a function is static (that is, private to the module being compiled) and all calls to that function are inlined, then there are no calls remaining that are not inline. Consequently, the compiler does not generate an out-of-line copy for the function, thus reducing the size of the resulting application.

If the address of the function is taken, it is possible that the function could be called through that derived pointer, so the compiler cannot guarantee that all calls have been accounted for. In such cases, an out-of-line copy will always be generated.

A function declared `inline` must be defined (its body must be included) in every file in which the function is used. This is normally done by placing the `inline` definition in a header file. Usually it is also declared `static`.

Inlining and Global `asm` Statements

Inlining imposes a particular ordering on functions. If functions A and B are both marked as `inline`, and each calls the other, only one of the `inline` qualifiers can be followed. Depending on which the compiler chooses to apply, either A will be generated with `inline` versions of B, or B will be generated with `inline` versions of A. Either case may result in no out-of-line copy of the inlined function being generated. The compiler reorders the functions within a module to get the best inlining result. Functionally, the code is the same, but this affects the resulting assembly file.

When global `asm` statements are used with the module, between the function definitions, the compiler cannot do this reordering process, because the `asm` statement might be affecting the behavior of the assembly code that is generated from the following C function definitions. Because of this, global `asm` statements can greatly reduce the compiler's ability to inline a function call.

Inlining and Sections

When inlining, any section directives or pragmas on the function definitions are ignored. For example,




```
section("secA") inline int add(int a, int b) { return a + b; }  
section("secB") int times_two(int a) { return add(a, a); }
```

Although `add()` and `times_two()` are to be generated into different code sections, this is ignored during the inlining process, so if the code for `add()` is inlined into `times_two()`, the inlined copy appears in section "secB" rather than section "secA". Only when out-of-line copies are generated (if necessary) does the compiler make use of any section directive or pragma applied to the out-of-line copy of the inlined function.

Inline Assembly Language Support Keyword (`asm`)

The `cc21k asm()` construct is used to code ADSP-21xxx assembly language instructions within a C/C++ function. The `asm()` construct is useful for expressing assembly language statements that cannot be expressed easily or efficiently with C/C++ constructs.

Using `asm()`, you can code complete assembly language instructions and specify the operands of the instruction using C/C++ expressions. When specifying operands with a C/C++ expression, you do not need to know which registers or memory locations contain C/C++ variables.

-  The compiler does not analyze code defined with the `asm()` construct; it passes this code directly to the assembler. The compiler does perform substitutions for operands of the formats `%0` through `%9`; however, it passes everything else through to the assembler without reading or analyzing it. This means that the compiler cannot apply any enabled workarounds for silicon errata that may be triggered either by the contents of the `asm` construct, or by the sequence of instructions formed by the `asm()` construct and the surrounding code produced by the compiler.
-  The `asm()` constructs are executable statements, and as such, may not appear before declarations within C/C++ functions. The `asm()` constructs may also be used at global scope, outside function declarations. Such `asm()` constructs are used to pass declarations and directives directly to the assembler. They are not executable constructs, and may not have any inputs or outputs, or affect any registers.
-  When optimizing, the compiler sometimes changes the order in which generated functions appear in the output assembly file. However, if global-scope `asm` constructs are placed between two function definitions, the compiler ensures that the function order is retained in the generated assembly file. Consequently, function inlining may be inhibited.

An `asm()` construct without operands takes the form as shown below.

```
asm("nop;");
```

The complete assembly language instruction, enclosed in quotes, is the argument to `asm()`.



The compiler generates a label before and after inline assembly instructions when generating debug code (the `-g` switch [on page 1-35](#)). These labels are used to generate the debug line information used by the debugger. If the inline assembler inserts conditionally assembled code, an undefined symbol error is likely to occur at link time. For example, the following code could cause undefined symbols if `MACRO` is undefined:

```
asm("#ifdef MACRO");  
asm(" // assembly statements");  
asm("#endif");
```

If the inline assembler changes the current section and thereby causes the compiler labels to be placed in another section, such as a data section (instead of the default code section), then the debug line information is incorrect for these lines.

Using `asm()` constructs with operands requires some additional syntax described in the following sections.

- [“asm\(\) Construct Syntax” on page 1-115](#)
- [“Assembly Construct Operand Description” on page 1-118](#)
- [“Assembly Constructs With Multiple Instructions” on page 1-124](#)
- [“Assembly Construct Reordering and Optimization” on page 1-125](#)
- [“Assembly Constructs With Input and Output Operands” on page 1-126](#)
- [“Assembly Constructs With Compile-Time Constants” on page 1-127](#)

- [“Assembly Constructs and Flow Control” on page 1-128](#)
- [“Guidelines on the Use of asm\(\) Statements” on page 1-129](#)

asm() Construct Syntax

Using `asm()` constructs, you can specify the operands of the assembly instruction using C/C++ expressions. You do not need to know which registers or memory locations contain C/C++ variables. Use the following general syntax for your `asm()` constructs.

```
asm [volatile] (
    template
    [:[constraint(output operand)[,constraint(output operand)...]]
    [:[constraint(input operand)[,constraint(input operand)...]]
    [:[clobber]]]
);
```

The syntax elements are defined as:

- **template**
The template is a string containing the assembly instruction(s) with `%number` indicating where the compiler should substitute the operands. Operands are numbered in order of appearance from left to right, starting at 0. Separate multiple instructions with a semicolon, and enclose the entire string within double quotes. For more information on templates containing multiple instructions, see [“Assembly Constructs With Multiple Instructions” on page 1-124](#).
- **constraint**
The constraint string directs the compiler to use certain groups of registers for the input and output operands. Enclose the constraint string within double quotes. For more information on operand constraints, see [“Assembly Construct Operand Description” on page 1-118](#).

- **output operand**

The output operands are the names of a C/C++ variables that receive output from corresponding operands in the assembly instructions.

- **input operand**

The input operand is a C/C++ expression that provides an input to a corresponding operand in the assembly instruction.

- **clobber**

The clobber notifies the compiler that a list of registers are overwritten by the assembly instructions. Use lowercase characters to name clobbered registers. Enclose each name within double quotes, and separate each quoted register name with a comma. The input and output operands are guaranteed not to use any of the clobbered registers, so you can read and write the clobbered registers as often as you like. See [Table 1-18 on page 1-123](#).

It is vital that any register overwritten by an assembly instruction and not allocated by the constraints is included in the clobber list. The list must include `memory` if an assembly instruction writes to memory.

asm() Construct Syntax Rules

These rules apply to assembly construct template syntax.

- The template is the only mandatory argument to `asm()`. All other arguments are optional.
- An operand constraint string followed by a C/C++ expression in parentheses describes each operand. For output operands, it must be possible to assign to the expression; that is, the expression must be legal on the left side of an assignment statement.

- A colon separates:
 - The template from the first output operand
 - The last output operand from the first input operand
 - The last input operand from the clobbered registers
- A space must be added between adjacent colon field delimiters in order to avoid a clash with the C++ “::” reserved global resolution operator.
- A comma separates operands and registers within arguments.
- The number of operands in arguments must match the number of operands in your template.
- The maximum permissible number of operands is ten (%0, %1, %2, %3, %4, %5, %6, %7, %8, and %9).



The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. The compiler does not parse the assembler instruction template, does not interpret the template, and does not verify whether the template contains valid input for the assembler.

asm() Construct Template Example

The following example shows how to apply the `asm()` construct template to the SHARC assembly language assignment instruction.

```
{
    int result, x;
    asm (
        "%0= %1;" :
        "=d" (result) :
        "d" (x)
    );
}
```

In the above example, note:

- The template is "`%0= %1;`". The `%0` is replaced with operand zero (`result`), the `%1` is replaced with operand one (`x`).
- The output operand is the C/C++ variable `result`. The letter `d` is the operand constraint for the variable. This constrains the output to a data register `R{0-15}`. The compiler generates code to copy the output from the `R` register to the variable `result`, if necessary. The `= in =d` indicates that the operand is an output.
- The input operand is the C/C++ variable `x`. The letter `d` in the operand constraint position for this variable constrains `x` to a data register `R{0-15}`. If `x` is stored in a different kind of register or in memory, the compiler generates code to copy the value into an `R` register before the `asm()` construct uses it.

Assembly Construct Operand Description

The second and third arguments to the `asm()` construct describe the operands in the assembly language template. Several pieces of information must be conveyed for the compiler to know how to assign registers to operands. This information is conveyed with an operand constraint. The compiler needs to know what kind of registers the assembly instructions can operate on, so it can allocate the correct register type.

You convey this information with a letter in the operand constraint string that describes the class of allowable registers.

[Table 1-17 on page 1-122](#) describes the correspondence between constraint letters and register classes.



The use of any letter not listed in [Table 1-17](#) results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

To assign registers to the operands, `cc21k` must also be informed which operands in an assembly language instruction are inputs, which are outputs, and which outputs may not overlap inputs.

The compiler is told this in three ways, such as:

- The output operand list appears as the first argument after the assembly language template. The list is separated from the assembly language template with a colon. The input operands are separated from the output operands with a colon and always follow the output operands.
- The operand constraints ([Table 1-17 on page 1-122](#)) describe which registers are modified by an assembly language instruction. The “=” in `=constraint` indicates that the operand is an output; all output operand constraints must use `=`. Operands that are input-outputs must use `+`. (See below.)
- The compiler may allocate an output operand in the same register as an unrelated input operand, unless the output or input operand has the `=&` constraint modifier. This is because `cc21k` assumes that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `=&` for each output operand that must not overlap an input.

Operand constraints indicate what kind of operand they describe by means of preceding symbols. The possible preceding symbols are: no symbol, `=`, `+`, `&`, `?`, and `#`.

- (no symbol)
The operand is an input. It must appear as part of the third argument to the `asm()` construct. The allocated register is loaded with the value of the C/C++ expression before the `asm()` template is exe-

cuted. Its C/C++ expression is not modified by the `asm()`, and its value may be a constant or literal.

Example: `d`

- **= symbol**

The operand is an output. It must appear as part of the second argument to the `asm()` construct. Once the `asm()` template has been executed, the value in the allocated register is stored into the location indicated by its C/C++ expression; therefore, the expression must be one that would be valid as the left-hand side of an assignment.

Example: `=d`

- **+ symbol**

The operand is both an input and an output. It must appear as part of the second argument to the `asm()` construct. The allocated register is loaded with the C/C++ expression value, the `asm()` template is executed, and then the allocated register's new value is stored back into the C/C++ expression. Therefore, as with pure outputs, the C/C++ expression must be one that is valid on the left-hand side of an assignment.

Example: `+d`

- **? symbol**

The operand is temporary. It must appear as part of the third argument to the `asm()` construct. A register is allocated as working space for the duration of the `asm()` template execution. The register's initial value is undefined, and the register's final value is discarded. The corresponding C/C++ expression is not loaded into the register, but must be present. This expression is normally specified using a literal zero.

Example: `?d`

- **& symbol**

This operand constraint may be applied to inputs and outputs. It indicates that the register allocated to the input (or output) may

not be one of the registers that are allocated to the outputs (or inputs). This operand constraint is used when one or more output registers are set while one or more inputs are still to be referenced. (This situation sometimes occurs if the `asm()` template contains more than one instruction.)

Example: `&d`

- **# symbol**

The operand is an input, but the register's value is clobbered by the `asm()` template execution. The compiler may make no assumptions about the register's final value. An input operand with this constraint will not be allocated the same register as any other input or output operand of the `asm()`. The operand must appear as part of the second argument to the `asm()` construct.

Example: `#d`

[Table 1-17](#) lists the registers that may be allocated for each register constraint letter. The use of any letter not listed in the “Constraint” column of this table results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter. [Table 1-18 on page 1-123](#) lists the registers that may be named as part of the clobber list

It is also possible to claim registers directly, instead of requesting a register from a certain class using the constraint letters. You can claim the registers directly by simply naming the register in the location where the class letter would be. The register names are the same as those used to specify the clobber list, as shown in [Table 1-18](#).

For example,

```
asm("%0 = %1 * %2;"
    : "=r13"(result)      /* output */
    : "r14"(x), "r15"(y)  /* input  */
    );
```

would load `x` into `r14`, load `y` into `r15`, execute the operation, and then store the total from `r13` back into `result`.



Naming the registers in this way allows the `asm()` construct to specify several registers that must be related, such as the DAG registers for a circular buffer. This also allows the use of registers not covered by the register classes accepted by the `asm()` construct. The clobber string can be any of the registers recognized by the compiler.

Table 1-17. `asm()` Operand Constraints

Constraint ¹	Register type	Registers
a	DAG2 B registers	b8 — b15
b	Q2 R registers	r4 — r7
c	Q3 R registers	r8 — r11
d	All R registers	r0 — r15
e	DAG2 L registers	l8 — l15
F	Floating-point registers	F0 — F15
f	Accumulator register	mrf, mrb
h	DAG1 B registers	b0 — b7
j	DAG1 L registers	l0 — l7
k	Q1 R registers	r0 - r3
l	Q4 R registers	r12 - r15
r	All general registers	r0 — r15, i0 — i15, l0 — l15, m0 — m15, b0 — b15, ustat1, ustat2
u	User registers	ustat1, ustat2 (also ustat3, ustat4 on ADSP-2116x, ADSP-2126x and ADSP-213xx processors)
w	DAG1 I registers	I0 — I7
x	DAG1 M registers	M0 — M7

Table 1-17. asm() Operand Constraints (Cont'd)

Constraint ¹	Register type	Registers
y	DAG2 I registers	I8 — I15
z	DAG2 M registers	M8 — M15
n	None (For more information, see “Assembly Constructs With Compile-Time Constants” on page 1-127.)	
=&constraint	Indicates that the constraint is applied to an output operand that may not overlap an input operand	
=constraint	Indicates that the constraint is applied to an output operand	
&constraint	Indicates the constraint is applied to an input operand that may not be overlapped with an output operand	
?constraint	Indicates the constraint is temporary	
+constraint	Indicates the constraint is both an input and output operand	
#constraint	Indicates that the constraint is an input operand whose value is changed	

- 1 The use of any letter not listed in [Table 1-17](#) results in unspecified behavior. The compiler does not check the validity of the code by using the constraint letter.

Table 1-18. Register Names for asm() Constructs

Clobber String	Meaning
"r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7", "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15"	General data registers
"i0", "i1", "i2", "i3", "i4", "i5", "i8", "i9", "i10", "i11", "i12", "i13", "i14", "i15"	Index registers
"m0", "m1", "m2", "m3", "m4", "m8", "m9", "m10", "m11", "m12"	Modifier registers
"b0", "b1", "b2", "b3", "b4", "b7", "b8", "b9", "b10", "b11", "b12", "b13", "b14", "b15",	Base registers

Table 1-18. Register Names for asm() Constructs (Cont'd)

Clobber String	Meaning
"r0", "r1", "r2", "r3", "r4", "r5", "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15"	Length registers
"mrf", "mrb"	Multiplier result registers
"acc", "mcc", "scc", "btf"	Condition registers
"lcntr"	Loop counter register
"PX"	PX register
"ustat1", "ustat2"	User-defined status registers
"memory"	Unspecified memory locations
The following registers are available on ADSP-2116x/2126x/2136x processors:	
"s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s10", "s11", "s12", "s13", "s14", "s15"	Shadow data registers
"smrf", "smrb"	Shadow multiplier result registers
"sacc", "smcc", "sscc", "sbtf"	Shadow condition registers
"ustat3", "ustat4"	User-defined status registers

Assembly Constructs With Multiple Instructions

There can be many assembly instructions in one template. Normal rules for line-breaking apply. In particular, the statement may spread over multiple lines. You are recommended not to split a string over more than one line, but to use the C language's string concatenation feature. If you are placing the inline assembly statement in a preprocessor macro, see [“Compound Macros” on page 1-265](#).

The following listing is an example of multiple instructions in a template.

```
/* (pseudo code) r7 = x; r6 = y; result = x + y; */
asm ("r7=%1;"
    "r6=%2;"
```

```

"%0=r6+r7;"
: "=d" (result)      /* output*/
: "d" (x), "d" (y)    /* input */
: "r7", "r6");        /* clobbers */

```

Do not attempt to produce multiple-instruction asm constructs via a sequence of single-instruction asm constructs, as the compiler is not guaranteed to maintain the ordering.

For example, the following should be avoided:

```

/* BAD EXAMPLE: Do not use sequences of single-instruction
** asms. Use a single multiple-instruction asm instead. */

asm("r7=%0;" : : "d" (x) : "r7");
asm("r6=%0;" : : "d" (y) : "r6");
asm("%0=r6+r7;" : "=d" (result));

```

Assembly Construct Reordering and Optimization

For the purpose of optimization, the compiler assumes that the side effects of an `asm()` construct are limited to changes in the output operands. This does not mean that you cannot use instructions with side effects, but be careful to notify the compiler that you are using them by using the clobber specifiers.

The compiler may eliminate supplied assembly instructions if the output operands are not used, move them out of loops, or reorder them with respect to other statements, where there is no visible data dependency. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

Use the keyword `volatile` to prevent an `asm()` instruction from being moved or deleted. For example,

```
asm volatile("idle;");
```

A sequence of `asm volatile()` constructs is not guaranteed to be completely consecutive; it may be moved across jump instructions or in other ways that are not significant to the compiler. To force the compiler to keep the output consecutive, use only one `asm volatile()` construct, or use the output of the `asm()` construct in a C/C++ statement.

Assembly Constructs With Input and Output Operands

When an `asm` construct has both inputs and outputs, there are two aspects to consider:

- Whether a value read from an input variable will be written back to the same variable or a different variable on output.
- Whether the input and output values will reside in the same register or different registers.

The most common case is when both input and output variables and input and output registers are different. In this case, the `asm` construct reads from one variable into a register, performs an operation which leaves the result in a different register, and writes that result from the register into a different output variable:

```
asm("%0 = %1;" : "=d" (newptr) : "d" (oldptr));
```

When the input and output variables are the same, it is usual that the input and output registers are also the same. In this case, you use the “+” constraint:

```
asm("%0 += 4;" : "+d" (sameptr));
```

When the input and output variables are different, but the input and output registers have to be the same (usually because of requirements of the assembly instructions), you indicate this to the compiler by using a different syntax for the input's constraint. Instead of specifying the register or class to be used, you specify the output to which the input must be matched.

For example,

```
asm("modify(%0,m7);"
    : "=w" (newptr)    // an output, given an I register,
                      // stored into newptr.
    : "0" (oldptr));   // an input, given same reg as %0,
                      // initialized from oldptr
```

This specifies that the input `oldptr` has `0` (zero) as its constraint string, which means it must be assigned the same register as `%0` (`newptr`).

Assembly Constructs With Compile-Time Constants

The `n` input constraint informs the compiler that the corresponding input operand should not have its value loaded into a register. Instead, the compiler is to evaluate the operand, and then insert the operand's value into the assembly command as a literal numeric value. The operand must be a compile-time constant expression. For example,

```
int r; int arr[100];
asm("%0 = %1;" : "=d" (r) : "d" (sizeof(arr))); // "d" constraint
```

produces code like

```
R0 = 100 (X);    // compiler loads value into register
R1 = R0;         // compiler replaces %1 with register
```

whereas:

```
int r; int arr[100];
asm("%0 = %1;" : "=d" (r) : "n" (sizeof(arr))); // "n" constraint
```

produces code like

```
R1 = 100;    // compiler replaces %1 with value
```

If the expression is not a compile-time constant, the compiler gives an error:

```
int r; int arr[100];
asm("%0 = %1;" : "=d" (r) : "n" (arr)); // error: operand
                                         // for "n" constraint
                                         // must be a compile-time constant
```

Assembly Constructs and Flow Control



Do not place flow control operations within an `asm()` construct that “leaves” the `asm()` construct functions, such as calling a procedure or performing a jump, to another piece of code that is not within the `asm()` construct itself. Such operations are invisible to the compiler, may result in multiple-defined symbols, and may violate assumptions made by the compiler.

For example, the compiler is careful to adhere to the calling conventions for preserved registers when making a procedure call. If an `asm()` construct calls a procedure, the `asm()` construct must also ensure that all conventions are obeyed, or the called procedure may corrupt the state used by the function containing the `asm()` construct.

It is also inadvisable to use labels in `asm()` statements, especially when function inlining is enabled. If a function containing such `asm` statements is inlined more than once in a file, there will be multiple definitions of the label, resulting in an assembler error. If possible, use PC-relative jumps in `asm` statements.

Guidelines on the Use of `asm()` Statements

There are certain operations that are performed more efficiently using other compiler features, and result in source code that is clearer and easier to read.

Accessing System Registers

System registers are accessed most efficiently using the functions in `sysreg.h` instead of using `asm()` statements. For example, the following `asm()` statement:

```
asm("R0 = 0; bit tst MODE1 IRPTEN; if TF r0 = r0 + 1; %0 = r0;"
    : "=d"(test) : : "r0");
```

can be written as:

```
#include <sysreg.h>
#include <def21060.h>
test = sysreg_bit_tst(sysreg_MODE1, IRPTEN);
```

Refer to [“Access to System Registers” on page 1-146](#) for more information.

Accessing Memory-Mapped Registers (MMRs)

MMRs can be accessed using the macros in the `Cdef*.h` files (for example, `Cdef21060.h`) that are supplied with VisualDSP++.

For example, `I0STAT` can be accessed using `asm()` statements, such as:

```
asm("R0 = 0x1234567; dm(I0STAT) = R0;" : : : "r0");
```

This can be written more cleanly and efficiently as:

```
#include <Cdef21060.h>
...
*pI0STAT = 0x1234567;
```

Dual Memory Support Keywords (pm dm)

This section describes `cc21k` language extension keywords to C and C++ that support the dual-memory space, modified Harvard architecture of the ADSP-21xxx processors. There are two keywords used to designate memory space: `dm` and `pm`. They can be used to specify the location of a static or global variable or to qualify a pointer declaration.

The following rules apply to dual memory support keywords:

- The memory space keyword (`dm` or `pm`) refers to the expression to the right of the keyword.
- You can specify a memory space for each level of pointer. This corresponds to one memory space for each `*` in the declaration.
- The compiler uses Data Memory (DM) as the default memory space for all variables. All undeclared spaces for data are Data Memory spaces.
- The compiler always uses Program Memory (PM) as the memory space for functions. Function pointers always point to Program Memory.
- You cannot assign memory spaces to automatic variables. All automatic variables reside on the stack, which is always in Data Memory.
- Literal character strings always reside in Data Memory.

The following listing shows examples of dual memory keyword syntax.

```
int pm buf[100];  
/* declares an array buf with 100 elements in Program Memory */  
int dm samples[100];  
/* declares an array samples with 100 elements in Data Memory */  
int points[100];  
/* declares an array points with 100 elements in Data Memory */  
int pm * pm xy;  
/* declares xy to be a pointer which resides in Program
```

```

    Memory and points to a Program Memory integer */
int dm * dm xy;
/* declares xy to be a pointer which resides in Data Memory and
   points to a Data Memory integer */
int *xy;
/* declares xy to be a pointer which resides in Data Memory
   and points to a Data Memory integer */
int pm * dm datp;
/* declares datp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */
int pm * datp;
/* declares datp to be a pointer which resides in Data Memory
   and points to a Program Memory integer */
int dm * pm progd;
/* declares progd to be a pointer which resides in Program
   Memory and points to a Data Memory integer */
int * pm progd;
/* declares progd to be a pointer which resides in Program
   Memory and points to a Data Memory integer */
float pm * dm * pm xp;
/* The first *xp is in Program Memory,
   following *xp in Data Memory, and xp itself is
   in Program Memory */

```

Memory space specification keywords cannot qualify type names and structure tags, but you can use them in pointer declarations. The following shows examples of memory space specification keywords in typedef and struct statements.

```

/* Dual Memory Support Keyword typedef & struct Examples */

typedef float pm * PFL0ATP;
    /* PFL0ATP defines a type which is a pointer to /
    /* a float which resides in pm.*/

struct s {int x; int y; int z;};
static pm struct s mystruct={10,9,8};
    /* Note that the pm specification is not used in */
    /* the structure definition. The pm specification */
    /* is used when defining the variable mystruct */

```

Memory Keywords and Assignments/Type Conversions

Memory space specifications limit the kinds of assignments your program can make, such as:

- You may make assignments between variables allocated in different memory spaces.
- Pointers to Program Memory must always point to Program Memory. Pointers to Data Memory must always point to Data Memory. You may not mix addresses from different memory spaces within one expression. Do not attempt to explicitly cast one type of pointer to another.

The following listings show a code segment with variables in different memory spaces being assigned and a code segment with illegal mixing of memory space assignments.

```
/* Legal Dual Memory Space Variable Assignment Example */
int pm x;
int dm y;
x = y;          /* Legal code */
```

```
/* Illegal Dual Memory Space Type Cast Example */
int pm *x;
int dm *y;
int dm a;
x = y;          /* Compiler will flag error */
x = &a;         /* Compiler will flag error */
```

Memory Keywords and Function Declarations/Pointers

Functions always reside in Program Memory. Pointers to functions always point to Program Memory. The following listing shows some sample function declarations with pointers.

```
/* Dual Memory Support Keyword Function Declaration (With Point-
ers) Syntax Examples */
int * y();          /* function y resides in    */
                   /* pm and returns a          */
                   /* pointer to an integer     */
                   /* which resides in dm       */
int pm * y();       /* function y resides in    */
                   /* pm and returns a          */
                   /* pointer to an integer     */
                   /* which resides in pm       */

int dm * y();       /* function y resides in    */
                   /* pm and returns a          */
                   /* pointer to an integer     */
                   /* which resides in dm       */

int * pm * y();     /* function y resides in    */
                   /* pm and returns a          */
                   /* pointer to a pointer      */
                   /* residing in pm that      */
                   /* points to an integer     */
                   /* which resides in dm       */
```

Memory Keywords and Function Arguments

The compiler checks calls to prototyped functions for memory space specifications consistent with the function prototype. The following listing shows sample code that `cc21k` flags as inconsistent use of memory spaces between a function prototype and a call to the function.

```
/* Illegal Dual Memory Support Keywords & Calls To Prototyped
Functions */
extern int foo(int pm*);
```

C/C++ Compiler Language Extensions

```
/* declare function foo() which expects a pointer to an int
residing in pm as its argument and which returns an int */

int x;          /* define int x in dm                                */

foo(&x);        /* call function foo()                                */
               /* using pm pointer (location of x) as the                */
               /* argument. cc21k FLAGS AS AN ERROR; this is an      */
               /* inconsistency between the function's              */
               /* declared memory space argument and function      */
               /* call memory space argument                      */
```

Memory Keywords and Macros

Using macros when making memory space specification for variables or pointers can make your code easier to maintain. If you must change the definition of a variable or pointer (moving it to another memory space), declarations that depend on the definition may need to be changed to ensure consistency between different declarations of the same variable or pointer.

To make changes of this type easier, you can use C/C++ preprocessor macros to define common memory spaces that must be coordinated. The following listing shows two code segments that are equivalent after pre-processing. The code segment guarded by `EASILY_CHANGED` lets you redefine the memory space specifications by redefining the macros `SPACE1` and `SPACE2`, and making it easy to redefine the memory space specifications at compile-time.

```
/* Dual Memory Support Keywords & Macros */

#ifdef EASILY_CHANGED
/* pm and dm can be easily changed at compile-time. */
#define SPACE1 pm
#define SPACE2 dm
char SPACE1 * foo (char SPACE2 *);
char SPACE1 * x;
```

```

char SPACE2 y;
x = foo(&y);

#else
/* not so easily changed. */
char pm * foo (char dm *);
char pm * x;
char dm y;
x = foo(&y);
#endif

```

Bank Type Qualifiers

Bank qualifiers can be attached to data declarations to indicate that the data resides in particular memory banks. For example,

```

int bank("blue") *ptr1;
int bank("green") *ptr2;

```

The bank qualifier assists the optimizer because the compiler assumes that if two data items are in different banks, they can be accessed together without conflict. The bank name string literals have no significance, except to differentiate between banks. There is no interpretation of the names attached to banks, which can be any arbitrary string. There is a current implementation limit of ten different banks.

For any given function, three banks are automatically defined. These are:

- The default bank for global data.
The “static” or “extern” data that is not explicitly placed into another bank is assumed to be within this bank. Normally, this bank is called “__data“, although a different bank can be selected with `#pragma data_bank(bankname)`.

- The default bank for local data.
Local variables of “auto” storage class that are not explicitly placed into another bank are assumed to be within this bank. Normally, this bank is called “__stack”, although a different bank can be selected with `#pragma stack_bank(bankname)`.
- The default bank for the function’s instructions.
The function itself is placed into this bank. Normally, it is called “__code”, although a different bank can be selected with `#pragma code_bank(bankname)`.

Each memory bank can have different performance characteristics. For more information on memory bank attributes, see [“Memory Bank Pragma” on page 1-210](#).

Placement Support Keyword (section)

The `section` keyword directs the compiler to place an object or function in an assembly `.SECTION` of the compiler’s intermediate output file. You name the assembly `.SECTION` directive with the `section()`’s string literal parameter. If you do not specify a `section()` for an object or function declaration, the compiler uses a default section. For information on the default sections, see [“Memory Usage” on page 1-270](#).

Applying `section()` is meaningful only when the data item is something that the compiler can place in the named section. Apply `section()` only to top-level, named objects that have a static duration, are explicitly `static`, or are given as external-object definitions.

The following example shows the declaration of a static variable that is placed in the section called `bingo`.

```
static section("bingo") int x;
```

The `section()` keyword has the limitation that section initialization qualifiers cannot be used within the section name string. The compiler may generate labels containing this string, which will result in assembly syntax

errors. Additionally, the keyword is not compatible with any pragmas that precede the object or function. For finer control over section placement and compatibility with other pragmas, use `#pragma section`.

Refer to “[#pragma section/#pragma default_section](#)” on page 1-201 for more information.



Note that `section` has replaced the `segment` keyword in earlier releases of the compiler. Although the `segment()` keyword is supported by the compiler of the current release, we recommend that you revise the legacy code.

Placement of Compiler-Generated Code and Data

If the `section()` keyword (“[Placement Support Keyword \(section\)](#)”) is not used, the compiler emits code and data into default sections. The `-section` switch ([on page 1-60](#)) can be used to specify alternatives for these defaults on the command-line, and the `default_section` pragma ([on page 1-201](#)) can be used to specify alternatives for some of them within the source file.

In addition, when using certain features of C/C++, the compiler may be required to produce internal data structures. The `-section` switch and the `default_section` pragma allow you to override the default location where the data would be placed. For example,

```
cc21k -section vtbl=vtbl_data test.cpp -c++
```

would instruct the compiler to place all the C++ virtual function look-up tables into the section `vtbl_data`, rather than the default `vtbl` section. It is the user’s responsibility to ensure that appropriately named sections exist in the `.ldf` file.

The compiler currently supports the following section identifiers:

C/C++ Compiler Language Extensions

<code>code</code>	Controls placement of machine instructions. Default is <code>seg_pmco</code> .
<code>data</code>	Controls placement of initialized variable data . Default is <code>seg_dmda</code>
<code>pm_data</code>	Controls placement of initialized data declared with the <code>_pm</code> keyword
<code>constdata</code>	Controls placement of constant data
<code>pm_constdata</code>	Controls placement of constant data declared with the <code>_pm</code> keyword
<code>bsz</code>	Controls placement of zero-initialized variable data,. Default is <code>.bss</code> .
<code>sti</code>	Controls placement of the static C++ class constructor “start” functions . Default is <code>seg_pmco</code> . For more information, see “Constructors and Destructors of Global Class Instances” on page 1-280.
<code>switch</code>	Controls placement of jump-tables used to implement C/C++ switch statements.
<code>strings</code>	Controls placement of string literals.
<code>vtbl</code>	Controls placement of the C++ virtual lookup tables. Default is <code>seg_vtbl</code> .
<code>vtable</code>	Synonym for <code>vtbl</code>
<code>autoinit</code>	Controls placement of data used to initialise aggregate autos
<code>alldata</code>	Controls placement of <code>data</code> , <code>constdata</code> , <code>bss</code> , <code>strings</code> and <code>autoinit</code> all at once

When both `-section` switches and `default_section` pragmas are used, the following priority is used:

1. A `default_section` pragma within the source has the highest priority.
2. The `-section` switch has precedence if no `default_section` pragma is in force.

Boolean Type Support Keywords (bool, true, false)

The `bool`, `true`, and `false` keywords are extensions that support the C++ boolean type. The `bool` keyword is a unique signed integral type, just as the `wchar_t` is a unique unsigned type. There are two built-in constants of this type: `true` and `false`. When converting a numeric or pointer value to `bool`, a zero value becomes `false` and a nonzero value becomes `true`. A `bool` value may be converted to `int` by promotion, taking `true` to one and `false` to zero. A numeric or pointer value is automatically converted to `bool` when needed.

These keyword extensions behave more or less as if the declaration that follows had appeared at the beginning of the file, except that assigning a nonzero integer to a `bool` type always causes it to take on the value `true`.

```
typedef enum { false, true } bool;
```

Pointer Class Support Keyword (restrict)

The `restrict` operator keyword is an extension that supports restricted pointer features. The use of `restrict` is limited to the declaration of a pointer and specifies that the pointer provides exclusive initial access to the object to which it points. More simply, `restrict` is a way that you can identify that a pointer does not create an alias. Also, two different restricted pointers cannot designate the same object, and therefore, they are not aliases.

The compiler is free to use the information about restricted pointers and aliasing to better optimize C/C++ code that uses pointers. The keyword is most useful when applied to function parameters about which the compiler would otherwise have little information.

For example,

```
void fir(short *in, short *c, short *restrict out, int n)
```

The behavior of a program is undefined if it contains an assignment between two restricted pointers, except for the following cases:

- A function with a restricted pointer parameter may be called with an argument that is a restricted pointer.
- A function may return the value of a restricted pointer that is local to the function, and the return value may then be assigned to another restricted pointer.

If you have a program that uses a restricted pointer in a way that it does not uniquely refer to storage, then the behavior of the program is undefined.

Variable-Length Array Support

The compiler supports variable-length automatic arrays when in C mode (Variable-length arrays are not supported for C++.) Unlike other automatic arrays, variable-length arrays are declared with a non-constant length. This means that the space is allocated when the array is declared, and deallocated when the brace-level is exited.

The compiler does not allow jumping into the brace-level of the array, and produces a compile time error message if this is attempted. The compiler does allow breaking or jumping out of the brace-level, and it deallocates the array when this occurs.

You can use variable-length arrays as function arguments, such as:

```
struct entry
tester (int len, char data[len][len])
{
    ...
}
```

The compiler calculates the length of an array at the time of allocation. It then remembers the array length until the brace-level is exited and can return it as the result of the `sizeof()` function performed on the array.

Because variable-length arrays must be stored on the stack, it is impossible to have variable-length arrays in Program Memory. The compiler issues an error if an attempt is made to use a variable-length array in `pm`.

As an example, if you were to implement a routine for computation of a product of three matrices, you need to allocate a temporary matrix of the same size as the input matrices. Declaring an automatic variable size matrix is much easier than explicitly allocating it in a heap.

The expression declares an array with a size that is computed at run time. The length of the array is computed on entry to the block and saved in case the `sizeof()` operator is used to determine the size of the array. For multidimensional arrays, the boundaries are also saved for address computation. After leaving the block, all the space allocated for the array is deallocated. For example, the following program prints 10, not 50.

```
main ()
{
    foo(10);
}

void foo (int n)
{
    char c[n];
    n = 50;
    printf("%d", sizeof(c));
}
```

Long Identifiers

The compiler supports identifiers of up to 1022 characters in length, or 1023 including name mangling.

Non-Constant Initializer Support

The `cc21k` compiler includes support for the ISO/ANSI standard definition of the C and C++ language and includes extended support for initializers. The compiler does not require the elements of an aggregate initializer for an automatic variable to be constant expressions.

The following example shows an initializer with elements that vary at run time.

```
void initializer (float a, float b)
{
    float the_array[2] = { a-b, a+b };
}

void foo (float f, float g)
{
    float beat_freqs[2] = { f-g, f+g };
}
```

Indexed Initializer Support

ANSI/ISO standard C/C++ requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized. The `cc21k` compiler C/C++, by comparison, supports labeling elements for array initializers. This feature lets you specify the array or structure elements in any order by specifying the array indices or structure field names to which they apply. All index values must be constant expressions, even in automatic arrays.

For an array initializer, the syntax `[INDEX]` appearing before an initializer element value specifies the index to be initialized by that value. Subsequent initializer elements are then applied to the sequentially following elements of the array, unless another use of the `[INDEX]` syntax appears. The index values must be constant expressions, even if the array being initialized is automatic.

The following example shows equivalent array initializers, the first in standard C/C++ and the next using the compiler's C/C++. Note that the [index] precedes the value being assigned to that element.

```
/* Example 1 Standard & cc21k C/C++ Array Initializer */
/* Standard array initializer */
int a[6] = { 0, 0, 15, 0, 29, 0 };

/* equivalent cc21k C/C++ array initializer */

int a[6] = { [4] 29, [2] 15 };
```

You can combine this technique of naming elements with standard C/C++ initialization of successive elements. The standard and cc21k instructions below are equivalent. Note that any unlabeled initial value is assigned to the next consecutive element of the structure or array.

```
/* Example 2 Standard & cc21k C/C++ Array Initializer */
/* Standard array initializer */

int a[6] = { 0, v1, v2, 0, v4, 0 };

/* equivalent cc21k C/C++ array initializer that uses */
/* indexed elements */

int a[6] = { [1] v1, v2, [4] v4 };
```

The following example shows how to label the array initializer elements when the indices are characters or an enum type.

```
/* Example 3 Array Initializer With enum Type Indices */
/* cc21k C/C++ array initializer */

int whitespace[256] =
{
    [' ' ] 1, ['\t'] 1, ['\v'] 1, ['\f'] 1, ['\n'] 1, ['\r'] 1
};

enum { e_ftp = 21, e_telnet = 23, e_smtp = 25, e_http = 80, e_nnntp
= 119 };
char *names[] = {
```

```
[e_ftp] "ftp",  
[e_http] "http",  
[e_nntp] "nntp",  
[e_smtp] "smtp",  
[e_telnet] "telnet"  
};
```

In a structure initializer, specify the name of a field to initialize with *fieldname*: before the element value. The standard C/C++ and cc21k C/C++ struct initializers in the example below are equivalent.

```
/* Example 4 Standard C & cc21k C/C++ struct Initializer */  
/* Standard C struct Initializer */
```

```
struct point {int x, y};  
struct point p = {xvalue, yvalue};
```

```
/* Equivalent cc21k C/C++ struct Initializer With  
Labeled Elements */
```

```
struct point {int x, y};  
struct point p = {y: yvalue, x: xvalue};
```

Aggregate Constructor Expression Support

Extended initializer support in cc21k C/C++ includes support for aggregate constructor expressions, which enable you to assign values to large structure types without requiring each element's value to be individually assigned.

The following example shows an ISO/ANSI standard C `struct` usage followed by equivalent cc21k C/C++ code that has been simplified using a constructor expression.

```
/* Standard C struct & cc21k C/C++ Constructor struct */  
/* Standard C struct */  
struct foo {int a; char b[2];};  
struct foo make_foo(int x, char *s)  
{
```



```

struct foo temp;
temp.a = x;
temp.b[0] = s[0];
if (s[0] != '\0')
    temp.b[1] = s[1];
else
    temp.b[1] = '\0';
return temp;
}

/* Equivalent cc21k C/C++ constructor struct */
struct foo make_foo(int x, char *s)
{
    return((struct foo) {x, {s[0], s[0] ? s[1] : '\0'}});
}

```

Preprocessor Generated Warnings

The preprocessor directive `#warning` causes the preprocessor to generate a warning and continue preprocessing. The text on the remainder of the line that follows `#warning` is used as the warning message.

C++ Style Comments

The compiler accepts C++ style comments in C programs, beginning with `//` and ending at the end of the line. This is essentially compatible with standard C, except for the following case.

```

a = b
/* highly unusual */ c
;

```

which a standard C compiler processes as:

```

a = b / c;

```

Compiler Built-In Functions

The compiler supports intrinsic (built-in) functions that enable efficient use of hardware resources. Knowledge of these functions is built into the cc21k compiler. Your program uses them via normal function call syntax. The compiler notices the invocation and generates one or more machine instructions, just as it does for normal operators, such as + and *.

Built-in functions have names which begin with `__builtin_`. Note that identifiers beginning with double underlines (`__`) are reserved by the C standard, so these names do not conflict with user program identifiers. The header files also define more readable names for the built-in functions without the `__builtin_` prefix. These additional names are disabled if the `-no-builtin` option is used.

This section describes:

- [“Access to System Registers” on page 1-146](#)
- [“Circular Buffer Built-In Functions” on page 1-151](#)
- [“Compiler Performance Built-In Functions” on page 1-152](#)
- [“Fractional Built-In Functions” on page 1-155](#)

The cc21k compiler provides built-in versions of some C library functions as described in section “Using Compiler Built-In C Library Functions” of the *VisualDSP++ 5.0 Run-Time Library Manual for SHARC Processors*.

Access to System Registers

The `sysreg.h` header file defines a set of functions that provide efficient system access to registers, modes, and addresses not normally accessible from C source. These functions are specific to individual architectures.

This section describes the functions that provide access to system registers. These functions are based on underlying hardware capabilities of the ADSP-21xxx processors. The functions are defined in the header file `sysreg.h`. They allow direct read and write access, as well as the testing and modifying of bit sets.

The functions are:

```
int sysreg_read (const int SR_number);
```

`sysreg_read` reads the value of the designated register and returns it.

```
void sysreg_write (const int SR_number, const int new_value);
```

`sysreg_write` stores the specified value in the nominated system register.

```
void sysreg_write_nop (const int SR_number, const int bit_mask);
```

`sysreg_write_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places a ‘NOP;’ after the instruction.

```
void sysreg_bit_clr (const int SR_number, const int bit_mask);
```

`sysreg_bit_clr` clears all the bits of the nominated system register that are set in the supplied bit mask.

```
void sysreg_bit_clr_nop (const int SR_number, const int bit_mask);
```

`sysreg_bit_clr_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places ‘NOP;’ after the instruction.

```
void sysreg_bit_set (const int SR_number, const int bit_mask);
```

`sysreg_bit_set` sets all the bits of the nominated system register that are also set in the supplied bit mask.

```
void sysreg_bit_set_nop (const int SR_number, const int bit_mask);
```

`sysreg_bit_set_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places 'NOP;' after the instruction.

```
void sysreg_bit_tgl (const int SR_number, const int bit_mask);
```

`sysreg_bit_tgl` toggles all the bits of the nominated system register that are set in the supplied bit mask.

```
void sysreg_bit_tgl_nop (const int SR_number, const int bit_mask);
```

`sysreg_bit_tgl_nop` toggles all the bits of the nominated system register that are set in the supplied bit mask, but also places 'NOP;' after the instruction.

```
int sysreg_bit_tst (const int SR_number, const int bit_mask);
```

`sysreg_bit_tst` returns a non-zero value if all of the bits that are set in the supplied bit mask are also set in the nominated system register.

```
int sysreg_bit_tst_all (const int SR_number, const int value);
```

`sysreg_bit_tst_all` returns a non-zero value if the contents of the nominated system register are equal to the supplied value.



The register names are defined in `sysreg.h` and must be a compile-time literal. The effect of using the incorrect function for the size of the register or using an undefined register number is undefined.

On all ADSP-21xxx processors, the system registers are:

<code>sysreg_IMASK</code>	<code>sysreg_IMASKP</code>
<code>sysreg_ASTAT</code>	<code>sysreg_STKY</code>
<code>sysreg_USTAT1</code>	<code>sysreg_USTAT2</code>

In addition, ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors have the following system registers:

<code>sysreg_LIRPTL</code>	<code>sysreg_MMASK</code>
<code>sysreg_ASTATY</code>	<code>sysreg_FLAGS</code>
<code>sysreg_STKYY</code>	<code>sysreg_USTAT3</code>
<code>sysreg_USTAT4</code>	

Due to hardware characteristics of the SHARC processors, the bit values for the `sysreg_bit_*` interrogation and manipulation functions must be compile-time constants. The following section provides the header filenames for the relevant processors.

- **ADSP-2106x and ADSP-21020 processors**
The header files `def21060.h`, `def21061.h`, `def210651.h` and `def21020.h` provide symbolic names for the individual bits in the system registers.
- **ADSP-2116x processors**
The header files `def21160.h` and `def21161.h` provide symbolic names for the individual bits in the system registers.
- **ADSP-2126x processors**
The header files `def21261.h`, `def21262.h`, `def21266.h` and `def21267.h` provide symbolic names for the individual bits in the system registers.
- **ADSP-2136x processors**
The header files `def21362.h`, `def21363.h`, `def21364.h`, `def21365.h`, `def21366.h`, `def21367.h`, `def21368.h` and `def21369.h` provide symbolic names for the individual bits in the system registers.
- **ADSP-2137x processors**
The header files `def21371.h` and `def21375.h` provide symbolic names for the individual bits in the system registers.
- **ADSP-2146x processors**
The header files `def21462.h`, `def21465.h`, `def21467.h`, and `def21469.h` provide symbolic names for the individual bits in the system registers.

Circular Buffer Built-In Functions

The C/C++ compiler provides the following two built-in functions for using the SHARC circular buffer mechanisms.

Circular Buffer Increment of an Index

The following operation performs a circular buffer increment of an index:

The equivalent operation is:

```
index +=incr;
if (index <0)index +=nitems;
else if (index >=nitems)index -=nitems;
```

Circular Buffer Increment of a Pointer

The following operation provides a circular buffer increment of an pointer.

```
int circptr(void * ptr, size_t incr, void * base, size_t buflen)
```

The equivalent operation is:

```
ptr +=incr;
if (ptr <base)ptr +=buflen;
else if (ptr >=(base+buflen))ptr -=buflen;
```

For more information on `circindex` and `circptr` library functions, refer to *VisualDSP++ 5.0 Run-Time Library Manual for SHARC Processors*.

The compiler also attempts to generate circular buffer increments for modulus array references, such as `array[index %nitems]`. For this to happen, the compiler must be able to determine that the starting value for `index` is within the range `0...(nitems-1)`. When the `-force-circbuf` switch (on page 1-34) is specified, the compiler always treats array references of the form `[i%n]` as a circular buffer operation on the array.

Compiler Performance Built-In Functions

Expected Behavior

The `expected_true` and `expected_false` functions provide the compiler with information about the expected behavior of the program. You can use these built-in functions to tell the compiler which parts of the program are most likely to be executed; the compiler can then arrange for the most common cases to be those that execute most efficiently.

```
#include <21060.h>
#include <210651.h>
#include <21160.h>
#include <21262.h>
#include <21363.h>
#include <21364.h>
#include <21365.h>
int expected_true(int cond);
int expected_false(int cond);
```

For example, consider the code

```
extern int func(int);
int example(int call_the_function, int value)
{
    int r = 0;
    if (call_the_function)
        r = func(value);
    return r;
}
```

If you expect that parameter `call_the_function` to be true in the majority of cases, you can write the function in the following manner:

```
extern int func(int);
int example(int call_the_function, int value)
{
    int r = 0;
    if (expected_true(call_the_function))
        // indicate most likely true
        r = func(value);
}
```



```

    return r;
}

```

This indicates to the compiler that you expect `call_the_function` to be true in most cases, so the compiler arranges for the default case to be to call function `func()`. If, on the other hand, you were to write the function as:

```

extern int func(int);
int example(int call_the_function, int value)
{
    int r = 0;
    if (expected_false(call_the_function))
        // indicate most likely false
        r = func(value);
    return r;
}

```

then the compiler arranges for the generated code to default to the opposite case, of not calling function `func()`.

These built-in functions do not change the operation of the generated code, which will still evaluate the boolean expression as normal. Instead, they indicate to the compiler which flow of control is most likely, helping the compiler to ensure that the most commonly-executed path is the one that uses the most efficient instruction sequence.

The `expected_true` and `expected_false` built-in functions only take effect when optimization is enabled in the compiler. They are only supported in conditional expressions.

Known Values

The `__builtin_assert()` function provides the compiler with information about the values of variables which it may not be able to deduce from the context. For example, consider the code

```

int example(int value, int loop_count)
{
    int r = 0;

```

C/C++ Compiler Language Extensions

```
int i;
for (i = 0; i < loop_count; i++) {
    r += value;
}
return r;
}
```

The compiler has no way of knowing what values may be passed in to the function. If you know that the loop count will always be greater than four, you can allow the optimizer to make use of that knowledge using

`__builtin_assert()`:

```
int example(int value, int loop_count)
{
    int r = 0;
    int i;
    __builtin_assert(loop_count > 4);
    for (i = 0; i < loop_count; i++) {
        r += value;
    }
    return r;
}
```

The optimizer can now omit the jump over the loop body it would otherwise have to emit to cover `loop_count == 0`. In more complicated code, further optimizations may be possible when bounds for variables are known.

Fractional Built-In Functions

The SHARC compiler provides a set of fractional built-in functions to support the C++ implementation provided in the header file. These built-in functions are:

- `float __builtin_conv_RtoF(int __a);`
Converts a fractional value to floating point representation. This function is implemented by a `float` instruction. Conversion from a fractional value to a floating-point value may result in some precision loss.
- `int __builtin_conv_FtoR(float __a);`
Converts a floating point value to a fractional representation. This function is implemented by a `fix` instruction and does not saturate. Conversion of a floating point value that cannot be represented as a fractional value will return `0xFFFFFFFF`.
- `int __builtin_RxR(int __a, int __b);`
Multiplies two fractional values, returning a fractional value. This function is implemented by a `multiply` instruction followed by a `sat` instruction. The function will saturate. The operation $(-1)*(-1)$ will return `0x7FFFFFFF`.
- `int __builtin_RxItoI(int __a, int __b);`
Multiplies a fractional value with an integral value, returning an integral value. This function is implemented as a `multiply` instruction followed by a `sat` instruction.
- `int __builtin_RxItoR(int __a, int __b);`
Multiplies a fractional value with an integral value, returning a fractional value. This function is implemented by a `multiply` instruction followed by a `sat` instruction. This function will saturate. Any negative number that cannot be represented by `fract` will return `0x80000000`, and any positive number that cannot be represented will return `0x7FFFFFFF`.

For additional fractional support for SHARC, refer to the section [“C++ Fractional Type Support” on page 1-228](#).

Miscellaneous Built-In Functions

int funcsize(const void *func)

The `funcsize` built-in function returns the size of function in bytes. The result is calculated from the difference between the start and end labels for the function operand. The compiler creates these labels for all C/C++ functions.

The start label is the mangled name of the function. The end label used is a dot (“.”) followed by the start label followed by “.end”. For example, for C function `foo`, these labels are “_foo:” and “_foo.end:”.

When using the `funcsize` built-in for assembly functions, the start and end labels need to be correctly defined for it to work.



The `funcsize` built-in does not work for functions defined in different modules than it is used, because end labels are not usually externally visible.

Pragmas

The compiler supports a number of pragmas. Pragmas are implementation-specific directives that modify the compiler’s behavior. There are two types of pragma usage: pragma directives and pragma operators.

Pragma directives have the following syntax:

```
#pragma pragma-directive pragma-directive-operands new-line
```

Pragma operators have the following syntax:

```
_Pragma ( string-literal )
```

When processing a pragma operator, the compiler effectively turns it into a pragma directive using a non-string version of *string-literal*. This means that the following pragma directive

```
#pragma linkage_name mylinkname
```

can also be equivalently be expressed using the following pragma operator

```
_Pragma ("linkage_name mylinkname")
```

The examples in this manual use the directive form.

The C compiler supports pragmas for:

- Arranging alignment of data
- Defining functions that can act as interrupt handlers
- Changing the optimization level, midway through a module
- Changing how an externally visible function is linked
- Header file configurations and properties
- Giving additional information about loop usage to improve optimizations

The following sections describe the pragmas that support these features:

- [“Data Alignment Pragmas” on page 1-158](#)
- [“Interrupt Handler Pragmas” on page 1-162](#)
- [“Loop Optimization Pragmas” on page 1-166](#)
- [“General Optimization Pragmas” on page 1-171](#)
- [“Function Side-Effect Pragmas” on page 1-173](#)
- [“Class Conversion Optimization Pragmas” on page 1-184](#)

- “Template Instantiation Pragas” on page 1-187
- “Header File Control Pragas” on page 1-189
- “Inline Control Pragas” on page 1-192
- “Linking Control Pragas” on page 1-195
- “Diagnostic Control Pragas” on page 1-207
- “Code Generation Pragas” on page 1-217
- “Exceptions Table Pragma” on page 1-218

The compiler issues a warning when it encounters an unrecognized pragma directive or pragma operator. Refer to Chapter 2, “[Achieving Optimal Performance from C/C++ Source Code](#)”, on how to use pragmas for code optimization.

Data Alignment Pragas

The data alignment pragmas include `align`, `alignment_region`, `pack` and `pad` pragmas. Alignments specified using these pragmas must be a power of two. The compiler rejects uses of those pragmas that specify alignments that are not powers of 2.

`#pragma align num`

This pragma may be used before variable declarations and field declarations. It applies to the variable or field declaration that immediately follows the pragma.

The pragma's effect is that the next variable or field declaration should be forced to be aligned on a boundary specified by *num*.

- If the pragma is being applied to a local variable (since local variables are stored on the stack), the alignment of the variable will only be changed when *num* is not greater than the stack alignment (that is, 2 words). If *num* is greater than the stack alignment, then a warning is given that the pragma is being ignored.
- If *num* is greater than the alignment normally required by the following variable or field declaration, then the variable or field declaration's alignment is changed to *num*.
- If *num* is less than the alignment normally required, then the variable or field declaration's alignment is changed to *num*, and a warning is given that the alignment has been reduced.

For example,

```
typedef struct {
#pragma align 4
    int foo;
    int bar;

#pragma align 4
    int baz;
} aligned_ints;
```

The pragma also allows the following keywords as allowable alignment specifications:

- `_WORD` – Specifies a 32-bit alignment
- `_LONG` – Specifies a 64-bit alignment
- `_QUAD` – Specifies a 128-bit alignment



The `align` pragma only applies to the immediately-following definition, even if that definition is part of a list. For example,

```
#pragma align 8
```

C/C++ Compiler Language Extensions

```
int i1, i2, i3;           // pragma only applies to i1
```

#pragma alignment_region (*alignopt*)

Sometimes it is desirable to specify an alignment for a number of consecutive data items rather than individually. This can be done using the `alignment_region` and `alignment_region_end` pragmas:

- `#pragma alignment_region` sets the alignment for all following data symbols up to the corresponding `alignment_region_end` pragma.
- `#pragma alignment_region_end` removes the effect of the active alignment region and restores the default alignment rules for data symbols.

The rules concerning the argument are the same as for `#pragma align`. The compiler faults an invalid alignment (such as an alignment that is not a power of two). The compiler warns if the alignment of a data symbol within the control of an `alignment_region` is reduced below its natural alignment (as for `#pragma align`).

Use of the `align` pragma overrides the region alignment specified by the currently active `alignment_region` pragma (if there is one). The currently active `alignment_region` does not affect the alignment of fields.

Example:

```
#pragma align 4

int aa;           /* alignment 4 */
int bb;           /* alignment 1 */

#pragma alignment_region (2)

int cc;           /* alignment 2 */
int dd;           /* alignment 2 */
int ee;           /* alignment 2 */
```



```

#pragma align 4

int ff;          /* alignment 4 */
int gg;          /* alignment 2 */
int hh;          /* alignment 2 */

#pragma alignment_region_end

int ii;          /* alignment 1 */

#pragma alignment_region (3)

long double kk;  /* the compiler faults this, alignment is not
                  a power of two */

#pragma alignment_region_end

```

#pragma pack (*alignopt*)

This pragma may be applied to struct definitions. It applies to all struct definitions that follow, until the default alignment is restored by omitting *alignopt*; for example, by `#pragma pack()` with empty parentheses.


The pragma is used to reduce the default alignment of the struct to be aligned. If there are fields within the struct that have a default alignment greater than *align*, their alignment is reduced to be *alignopt*. If there are fields within the struct that have alignment less than *align*, their alignment is unchanged.

If *alignopt* is specified, it is illegal to invoke `#pragma pad` until the default alignment is restored. The compiler generates an error if the `pad` and `pack` pragmas are used in a manner that conflicts.

#pragma pad (*alignopt*)

This pragma may be applied to `struct` definitions. It applies to all `struct` definitions that follow, until the default alignment is restored by omitting *alignopt*. This pragma is effectively shorthand for placing `#pragma align` before every field within the `struct` definition. Like the `#pragma pack`, it reduces the alignment of fields that default to an alignment greater than *alignopt*. However, unlike the `#pragma pack`, it also increases the alignment of fields that default to an alignment less than *alignopt*.

If *alignopt* is specified, it is illegal to invoke `#pragma pad` until default alignment is restored.

 While `#pragma pack (alignopt)` generates a warning if a field alignment is reduced, `#pragma pad (alignopt)` does not. If *alignopt* is specified, it is illegal to invoke `#pragma pack`, until default alignment is restored.

The following example shows how to use `#pragma pad()`.

```
#pragma pad(4)
struct {
    int i;
    int j;
} s = {1,2};
#pragma pad()
```

Interrupt Handler Pragas

The interrupt pragmas provide a method by which the user can write interrupt service routines in C and install them directly into the interrupt vector table, bypassing the dispatcher provided with the C run-time library.

#pragma implicit_push_sts_handler

When this pragma is applied to an interrupt handler, the compiler does not generate an explicit `PUSH` and `POP` of `STS`. This pragma applies only when compiling for ADSP-211xx, ADSP-212xx, ADSP-213xx, and ADSP-214xx SHARC processors. When compiling for pre-ADSP-211xx SHARC processors, the pragma is silently ignored.

The pragma takes effect only when it is used in conjunction with one of the interrupt pragmas for ADSP-211xx, ADSP-212xx, ADSP-213xx, and ADSP-214xx SHARC processors (for example, `interrupt_complete`), otherwise an error message will be issued.

The compiler can't determine whether the handler for the pragma is applied as a handler for the `VIRPT`, `IRQ`, or timer interrupts. The user must determine whether or not the pragma should be used.

#pragma interrupt

The `#pragma interrupt` pragma may be used before a function declaration or definition. It applies to the function declaration or definition that immediately follows this pragma. Use of this pragma causes the compiler to generate the function code so that it may be used as a self dispatching interrupt handler.

The `interrupt` pragma indicates that the compiler should ensure that all used registers (including scratch registers) are restored at the end of the function. The compiler also ensures that, if an `I` register is used in the function, the corresponding `L` register is set to zero, so that circular buffering is not inadvertently invoked.

The `#pragma interrupt` pragma should be used for interrupt handlers that are enabled with the `interruptss()` or `signalss()` family of interrupt functions, as these functions ensure that the correct dispatcher is called. For maximum benefit, `#pragma interrupt` should be used for leaf routines only (that is, functions which do not call any other functions). This is because the interrupt handler ensures that `L` registers are zeroed for the `I`

registers used in the function. If a function call is present, the handler must ensure that all appropriate L registers are set to zero. This adds considerably to the execution time of the handler.

#pragma interrupt_complete_nesting

The `#pragma interrupt_complete_nesting` pragma is used before a function definition, which is to be used as an interrupt handler that can be called directly from the interrupt vector table. Like `#pragma interrupt`, it saves and restores all registers used by the function. However, on the ADSP-211xx, ADSP-212xx, and ADSP-213xx processors, it also performs a `PUSH STS` instruction at the start of the function to save the status and `MODE1` registers.

Since this instruction disables nested interrupts, and this pragma can be used with nested interrupts, it re-enables interrupts by way of the `BIT SET MODE1 0x1000;` instruction. At the end of the function, it performs a `POP STS` instruction to restore the status and `MODE1` registers.

On ADSP-2106x processors, there is not enough space on the status stack to perform the `PUSH STS` and `POP STS` instructions so the compiler generates code that (apart from storing and restoring all the registers used by the function) also does the following:

- At the start of the function, `MODE1` and `ASTAT` are stored on the `MODE1` register. `BR0` and `BR8` are cleared with `RND32` being set. These are the default settings for these registers used by the compiler and libraries.
- Note that the `FLAGS` registers and `ASTAT` are located on the same register. At the end of the function, the compiler generates the following code to ensure that any changes to the `FLAGS` registers are preserved:
 1. Reads the new `ASTAT`
 2. Reads the `FLAGS` registers from it

3. Reads the original `ASTAT`
4. Clears the `FLAGS` registers on it
5. ORs the new `FLAGS` registers onto the original `ASTAT`

#pragma interrupt_complete

The `#pragma interrupt_complete` pragma is similar to the `#pragma interrupt_complete_nesting` pragma, except that it does not re-enable interrupts. (It is for non-nested interrupt handlers.) On the ADSP-211xx, ADSP-212xx, and ADSP-213xx processors, this is done by not modifying the `MODE1` register. On the ADSP-2106x processor, this is done by disabling interrupts at the start of the function, and then re-enabling them at the end of the function.

Interrupt Pragas and the Interrupt Vector Table

Since the interrupt handlers created by the `#pragma interrupt_complete_nesting` and `#pragma interrupt_complete` pragmas are called directly from the interrupt vector table, the calls to these handlers have to be placed directly into the interrupt vector table. For example, if the following interrupt handler is defined in this code:

```
#pragma interrupt_complete_nesting
void foo(void) {
    ....
}
```

Then for the handler `foo` to be called, the `crt` file must be modified.
Change the code:

```
___lib_SFT0I:    INT(USR0);
```

to

```
___lib_SFT0I:    jump(PC, _foo); nop; nop; nop;
```

This modified `crt` file must be included in the project so that it is built and linked in.



For more information on using interrupts, see [“Support for Interrupts” on page 1-247](#).

Loop Optimization Pragmas

Loop optimization pragmas give the compiler additional information about usage within a particular loop, which allows the compiler to perform more aggressive optimization. The pragmas are placed before the loop statement, and apply to the statement that immediately follows, which must be a `for`, `while` or `do` statement to have effect. In general, it is most effective to apply loop pragmas to inner-most loops, since the compiler can achieve the most savings there.

The optimizer always attempts to vectorize loops when it is safe to do so. The optimizer exploits the information generated by the interprocedural analysis ([“Interprocedural Analysis” on page 1-84](#)) to increase the cases where it knows it is safe to do so. Consider the following code:

#pragma SIMD_for

This pragma is used with ADSP-2116x, ADSP-2126x, ADSP-213xx, and ADSP-2146x processors. It enables the compiler to take advantage of Single-Instruction Multiple-Data (SIMD) operations. See [“SIMD Support” on page 1-231](#) for more details.

#pragma all_aligned

This pragma applies to the subsequent loop. This pragma tells that compiler that all pointer-induction variables in the loop are initially aligned to the maximum permitted alignment of the processor architecture. For ADSP-2106x processors, it is word-aligned; for ADSP-2116x, ADSP-2126x and ADSP-213xx processors, it is double-word aligned.

The variable takes an optional argument (*n*) which can specify that the pointers are aligned after *n* iterations. Therefore, `#pragma all_aligned(1)` says that after one iteration, all the pointer induction variables of the loop are so aligned. In other words, the default argument is zero.

#pragma no_vectorization

This pragma is used with ADSP-2116x, ADSP-2126x and ADSP-213xx processors. It ensures the compiler does not generate vectorized SIMD code for the loop on which it is specified.

#pragma loop_count (*min*, *max*, *modulo*)

This pragma appears just before the loop it describes. It asserts that the loop iterates at least *min* times, no more than *max* times, and a multiple of *modulo* times. This information enables the optimizer to omit loop guards and to decide whether the loop is worth completely unrolling and whether code needs to be generated for odd iterations. Any of the parameters of the pragma that are unknown may be left blank.

For example,

```
int i;
#pragma loop_count(24, 48, 8)
for (i=0; i < n; i++)
```

#pragma loop_unroll *N*

The `loop_unroll` pragma can be used only before a `for`, `while` or `do..while` loop. The pragma takes exactly one positive integer argument, *N*, and it instructs the compiler to unroll the loop *N* times prior to further transforming the code.

In the most general case, the effect of:

```
#pragma loop_unroll N
for ( init statements; condition; increment code) {
    loop_body
}
```

is equivalent to transforming the loop to:

```
for ( init statements; condition; increment code) {
    loop_body    /* copy 1 */
    increment_code
    if (!condition)
        break;

    loop_body    /* copy 2 */
    increment_code
    if (!condition)
        break;

    ...

    loop_body    /* copy N-1 */
    increment_code
    if (!condition)
        break;

    loop_body    /* copy N */
}
```


Similarly, the effect of

```
#pragma loop_unroll N
while ( condition ) {
    loop_body
}
```

is equivalent to transforming the loop to:

```
while ( condition ) {
    loop_body /* copy 1 */
    if (!condition)
        break;

    loop_body /* copy 2 */
    if (!condition)
        break;

    ...

    loop_body /* copy N-1 */
    if (!condition)
        break;

    loop_body /* copy N */
}
```

and the effect of:

```
#pragma loop_unroll N
do {
    loop_body
} while ( condition )
```

is equivalent to transforming the loop to:

```
do {
    loop_body /* copy 1 */
    if (!condition)
        break;
```

C/C++ Compiler Language Extensions

```
    loop_body    /* copy 2 */  
    if (!condition)  
        break;  
  
    ...  
  
    loop_body    /* copy N-1 */  
    if (!condition)  
        break;  
  
    loop_body    /* copy N */  
} while ( condition )
```

#pragma no_alias

Use this pragma to tell the compiler that the following loop has no loads or stores that conflict. When the compiler finds memory accesses that potentially refer to the same location through different pointers, known as “aliases”, the compiler is restricted in how it may reorder or vectorize the loop, because all the accesses from earlier iterations must be complete before the compiler can arrange for the next iteration to start.

In the example,

```
void vadd(int *a, int *b, int *out, int n) {  
    int i;  
    #pragma no_alias  
    for (i=0; i < n; i++)  
        out[i] = a[i] + b[i];  
}
```

the use of `no_alias` pragma just before the loop informs the compiler that the pointers `a`, `b` and `out` point to different arrays, so no load from `b` or `a` is using the same address as any store to `out`. Therefore, `a[i]` or `b[i]` is never an alias for `out[i]`.

Using the `no_alias` pragma can lead to better code because it allows the loads and stores to be reordered and any number of iterations to be performed concurrently (rather than just two at a time), thus providing better software pipelining by the optimizer.

#pragma vector_for

This pragma tells the compiler that all iterations of the loop may be run in parallel with each other. The `vector_for` pragma does not force the compiler to vectorize the loop. The optimizer checks various properties of the loop and does not vectorize it if it believes it is unsafe or if it cannot deduce that the various properties necessary for the vectorization transformation are valid.

Strictly speaking, the pragma simply disables checking for loop-carried dependencies.

```
void copy(short *a, short *b) {
    int i;
    #pragma vector_for
        for (i=0; i<100; i++)
            a[i] = b[i];
}
```

In cases where vectorization is impossible (for example, if array `a` were aligned on a word boundary but array `b` was not), the information given in the assertion made by `vector_for` may still be put to good use in aiding other optimizations.

General Optimization Pragmas

The compiler supports several pragmas which can change the optimization level while a given module is being compiled. These pragmas must be used globally, immediately prior to a function definition. The pragmas do not

just apply to the immediately following function; they remain in effect until the end of the compilation, or until superseded by one of the following `optimize_` pragmas.

- `#pragma optimize_off`
This pragma turns off the optimizer, if it was enabled, meaning it has the same effect as compiling with no optimization enabled.
- `#pragma optimize_for_space`
This pragma turns the optimizer back on, if it was disabled, or sets the focus to give *reduced code size* a higher priority than high performance, where these conflict.
- `#pragma optimize_for_speed`
This pragma turns the optimizer back on, if it was disabled, or sets the focus to give *high performance* a higher priority than reduced code size, where these conflict.
- `#pragma optimize_as_cmd_line`
This pragma resets the optimization settings to be those specified on the `cc21k` command line when the compiler was invoked.

These are code examples for the `optimize_` pragmas.

```
#pragma optimize_off
void non_op() { /* non-optimized code */ }

#pragma optimize_for_space
void op_for_si() { /* code optimized for size */ }

#pragma optimize_for_speed
void op_for_sp() { /* code optimized for speed */ }
/* subsequent functions declarations optimized for speed */
```

Function Side-Effect Pragmas

The function side-effect pragmas (`alloc`, `pure`, `const`, `regs_clobbered`, `overlay` and `result_alignment`) are used before a function declaration to give the compiler additional information about the function in order to enable it to improve the code surrounding the function call. These pragmas should be placed before a function declaration and should apply to that function. For example,

```
#pragma pure
long dot(short*, short*, int);
```

#pragma alloc

The `alloc` pragma tells the compiler that the function behaves like the library function “`malloc`”, returning a pointer to a newly allocated object. An important property of these functions is that the pointer returned by the function does not point at any other object in the context of the call. In the example,

```
#define N 100

#pragma alloc
int *new_buf(void);
int *vmul(int *a, int *b) {
    int i;
    int *out = new_buf();
    for (i = 0; i < N; ++i)
        out[i] = a[i] * b[i];
    return out;
}
```

the compiler can reorder the iterations of the loop because the `#pragma alloc` tells it that `a` and `b` cannot overlap `out`.

The GNU attribute `malloc` is also supported with the same meaning.

#pragma const

The `const` pragma is a more restrictive form of the `pure` pragma. The pragma tells the compiler that the function does not read from global variables as well as not writing to them or reading or writing volatile variables. The result of the function is therefore a function of its parameters. If any of the parameters are pointers, the function may not even read the data they point at.

#pragma misra_func(arg)

The `misra_func(arg)` pragma is placed before a function prototype. It is used to support MISRA-C rules 20.4, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12. The `arg` indicates the type of function with respect to the MISRA-C Rule. Functions following Rule 20.4 would take `arg heap`, 20.7 `arg jmp`, 20.8 `arg handler`, 20.9 `arg io`, 20.10 `arg string_conv`, 20.11 `arg system` and 20.12 `arg time`.

#pragma noreturn

The `noreturn` pragma can be placed before a function prototype or definition. The pragma tells the compiler that the function to which it applies will never return to its caller. For example, a function such as the standard C function “`exit`” never returns.

The use of this pragma allows the compiler to treat all code following a call to a function declared with the pragma as unreachable and hence removable.

```
#pragma noreturn
void func() {
    while(1);
}

main() {
    func();
    /* any code here will be removed */
}
```

#pragma pgo_ignore

The `pgo_ignore` pragma tells the compiler that no profile should be generated for this function, when using profile-guided optimization. This is useful when the function is concerned with error checking or diagnostics.

```
extern const short *x, *y;
int dotprod(void) {
    int i, sum = 0;
    for (i = 0; i < 100; i++)
        sum += x[i] * y[i];
    return sum;
}

#pragma pgo_ignore
int check_dotprod(void) {
    /* The compiler will not profile this comparison */
    return dotprod() == 100;
}
```

#pragma pure

This pragma tells the compiler that the function does not write to any global variables, and does not read or write any volatile variables. Its result, therefore, is a function of its parameters or of global variables. If any of the parameters are pointers, the function may read the data they point at but it may not write it.

As this means the function call has the same effect every time it is called, between assignments to global variables, the compiler does not need to generate the code for every call.

Therefore, in this example,

```
#pragma pure
long sdot(short *, short *, int);

long tendots(short *a, short *b, int n) {
    int i;
```

C/C++ Compiler Language Extensions

```
long s = 0;
for (i = 1; i < 10; ++i)
    s += sdot(a, b, n); // call can get hoisted out of loop
return s;}
```

the compiler can replace the ten calls to `sdot` with a single call made before the loop.

#pragma regs_clobbered *string*

This pragma may be used with a function declaration or definition to specify which registers are modified (or clobbered) by that function. The *string* contains a list of registers and is case-insensitive.

When used with an external function declaration, this pragma acts as an assertion telling the compiler something it would not be able to discover for itself. In the example,

```
#pragma regs_clobbered "r4 r8 i4"
void f(void);
```

the compiler knows that only registers `r4`, `r8` and `i4` may be modified by the call to `f`, so it may keep local variables in other registers across that call.


The `regs_clobbered` pragma may also be used with a function definition, or a declaration preceding a definition (when it acts as a command to the compiler to generate register saves, and restores on entry and exit from the function) to ensure it only modifies the registers in *string*.

For example,

```
#pragma regs_clobbered "r3 m4 r5 i12"
// Function "g" will only clobber r3, m4, r5, and i12
int g(int a) {
```




```
    return a+3;
}
```

 The `regs_clobbered` pragma may not be used in conjunction with `#pragma interrupt`. If both are specified, a warning is issued and the `regs_clobbered` pragma is ignored.

To obtain optimum results with the pragma, it is best to restrict the clobbered set to be a subset of the default scratch registers. When considering when to apply the `regs_clobbered` pragma, it may be useful to look at the output of the compiler to see how many scratch registers were used.

Restricting the volatile set to these registers will produce no impact on the code produced for the function but may free up registers for the caller to allocate across the call site.

 The `regs_clobbered` pragma cannot be used in any way with pointers to functions. A function pointer cannot be declared to have a customized clobber set, and it cannot take the address of a function which has a customized clobber set. The compiler raises an error if either of these actions are attempted.

String Syntax

A `regs_clobbered` string consists of a list of registers, register ranges, or register sets that are clobbered (Table 1-19). The list is separated by spaces, commas, or semicolons.

A *register* is a single register name, which is the same as that which may be used in an assembly file.

A *register range* consists of `start` and `end` registers which both reside in the same register class, separated by a hyphen. All registers between the two (inclusive) are clobbered.

A *register set* is a name for a specific set of commonly clobbered registers that is predefined by the compiler. Table 1-19 shows defined clobbered register sets,

Table 1-19. Clobbered Register Sets

Set	Registers
CCset	ASTAT, ASTATy (ADSP-2116x and ADSP-2126x processors only)
SHADOWset	All S regs, all Shadow MR regs, ASTATy. Always clobbered whether specified or not.
MRset	MRF, MRB; shadow MRF, shadow MRB (ADSP-2116x and ADSP-2126x processors only)
DAG1scratch	Members of DAG1 I, M, B and L-registers that are scratch by default
DAG2scratch	Members of DAG2 I, M, B and L-registers that are scratch by default
DAGscratch	All members of DAG1scratch and DAG2scratch
Dscratch	All D-registers that are scratch by default, ASTAT
ALLscratch	Entire default scratch register set

When the compiler detects an illegal string, a warning is issued and the default volatile set as defined in this compiler manual is used instead.

Unclobberable and Must Clobber Registers

There are certain caveats as to what registers may or must be placed in the clobbered set ([Table 1-19](#)). On SHARC processors, certain registers may not be specified in the clobbered set, as the correct operation of the function call requires their value to be preserved.

If the user specifies these registers in the clobbered set, a warning is issued and they are removed from the specified clobbered set.

II6, I7, B6, B7, L6, L7

Registers from these classes,

D, I, B, USTAT, LCNTR, PX, MR

may be specified in the clobbered set and code is generated to save them as necessary.

The L-registers are required to be set to zero on entry and exit from a function. A user may specify that a function clobbers the L-registers. If it is a compiler-generated function, then it leaves the L-registers at zero at the end of the function. If it is an assembly function, then it may clobber the L-registers. In that case, the L-registers are re-zeroed after any call to that function. The soft-wired registers M5, M6, M7 and M13, M14, M15 are reset in an analogous manner.

The registers R2 and I12 are always clobbered. If the user specifies a function definition with the `regs_clobbered` pragma that does not contain these registers, a warning is issued and these registers are added to the clobbered set.

User-Reserved Registers

User-reserved registers, which are indicated via the `-reserve` switch (on [page 1-59](#)), are never preserved in the function wrappers whether in the clobbered set or not.

Function Parameters

Function calling conventions are visible to the caller and do not affect the clobbered set that may be used on a function. For example,

```
#pragma regs_clobbered ""// clobbers nothing
void f(int a, int b);
void g() {
    f(2,3);
}
```

The parameters `a` and `b` are passed in registers R4 and R8, respectively. No matter what happens in function `f`, after the call returns, the values of R4 and R8 are still set to 2 and 3, respectively.

Function Results

The registers in which a function returns its result must always be clobbered by the callee and retain their new value in the caller. They may appear in the clobbered set of the callee but it makes no difference to the generated code—the return register are not saved and restored. Only the

return register used by the particular function return type is special. Return registers used by different return types are treated in the clobbered list in the convention way.

For example,

```
typedef struct { int x, int y } Point;
typedef struct { int x[10] } Big;
int f(); // Result in R0. R1 may be preserved across call
Point g();// Result in R0 and R1
Big f(); // Result pointer in R0, R1 may be preserved
        across call.
```

#pragma regs_clobbered_call string

This pragma may be applied to a statement to indicate that the call within the statement uses a modified volatile register set. The pragma is closely related to `#pragma regs_clobbered`, but avoids some of the restrictions that relate to that pragma.

These restrictions arise because the `regs_clobbered` pragma applies to a function's declaration—when the call is made, the clobber set is retrieved from the declaration automatically. This is not possible when the declaration is not available, because the function being called is not directly tied to a declaration of a specific function. This affects:

- pointers to functions
- class methods
- pointers to class methods
- virtual functions

In such cases, the `regs_clobbered_call` pragma can be used at the call site to inform the compiler directly of the volatile register set to be used during the call.

The pragma's syntax is as follows:

```
#pragma regs_clobbered_call clobber_string
    statement
```

where *clobber_string* follows the same format as for the `regs_clobbered` pragma and *statement* is the C statement containing the call expression.

There must be only a single call within the statement; otherwise, the statement is ambiguous. For example,

```
#pragma regs_clobbered "r0 r1 r2 i12"
int func(int arg) { /* some code */ }

int (*fnptr)(int) = func;

int caller(int value) {
    int r;

    #pragma regs_clobbered_call "r0 r1 r2 i12"
    r = (*fnptr)(value);
    return r;
}
```



When you use the `regs_clobbered_call` pragma, you must ensure that the called function does indeed only modify the registers listed in the clobber set for the call—the compiler does not check this for you. It is valid for the callee to clobber less than is listed in the call's clobber set. It is also valid for the callee to modify registers outside of the call's clobber set, as long as the callee saves the values first and restores them before returning to the caller.

The following examples show this.

Example 1:

```
#pragma regs_clobbered "r0 r1 r2 i12"
void callee(void) { ... }
```

C/C++ Compiler Language Extensions

```
#pragma regs_clobbered_call "r0 r1 r2 i12"
callee();    // Okay - clobber sets match
```

Example 2:

```
#pragma regs_clobbered "r0 r2 i12"
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r1 r2 i12"
callee();    // Okay - callee clobber set is a subset
               // of call's set
```

Example 3:

```
#pragma regs_clobbered "r0 r1 r2 i12"
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r2 i12"
callee();    // Error - callee clobbers more than
               // indicated by call.
```

Example 4:

```
void callee(void) { ... }

#pragma regs_clobbered_call "r0 r1 r2 i12"
callee();    // Error - callee uses default set larger
               // than indicated by call.
```

Limitations

Pragma `regs_clobbered_call` may not be used on constructors or destructors of C++ classes.

The pragma only applies to the call in the immediately-following statement. If the immediately-following line contains more than one statement, the pragma only applies to the first statement on the line:

```
#pragma regs_clobbered "r0 r1 r2 i12"
x = foo(); y = bar(); // only "x = foo();" is affected by
                       // the pragma.
```

Similarly, if the immediately-following line is a sequence of declarations that use calls to initialize the variables, then only the first declaration is affected:

```
#pragma regs_clobbered "r0 r1 r2 i12"
int x = foo(), y = bar();// only "x = foo()" is affected
                        // by the pragma.
```

Moreover, if the declaration with the call-based initializer is not the first in the declaration list, the pragma will have no effect:

```
#pragma regs_clobbered "r0 r1 r2 i12"
int w = 4, x = foo(); y = bar();// pragma has no effect
                                // on "w = 4".
```

The pragma has no effect on function calls that get inlined. Once a function call is inlined, the inlined code obeys the clobber set of the function into which it has been inlined. It does not continue to obey the clobber set that will be used if an out-of-line copy is required.

#pragma overlay

When compiling code which involves one function calling another in the same source file, the compiler optimizer can propagate register information between the functions. This means that it can record which scratch registers are clobbered over the function call. This can cause problems when compiling overlaid functions, as the compiler may assume that certain scratch registers are not clobbered over the function call, but they are clobbered by the overlay manager. The `#pragma overlay`, when placed on the definition of a function, will disable this propagation of register information to the function's callers.

For example,

```
#pragma overlay
int add(int a, int b)
{
    // callers of function add() assume it clobbers
    // all scratch registers
```

```
    return a+b;
}
```

#pragma result_alignment (*n*)

This pragma asserts that the pointer or integer returned by the function has a value that is a multiple of *n*.

The pragma is often used in conjunction with the `#pragma alloc` of custom-allocation functions that return pointers that are more strictly aligned than could be deduced from their type. The following example shows a use of the pragma. Note that this pragma will not change the alignment of data returned by the declared function. It is a guideline to the compiler.

```
#pragma result_alignment(8)
int * alloc_align8_data(unsigned long size);
```

Class Conversion Optimization Pragmas

The class conversion optimization pragmas (`param_never_null`, `suppress_null_check`) allow the compiler to generate more efficient code when converting class pointers from a pointer-to-derived-class to a pointer-to-base-class, by asserting that the pointer to be converted will never be a null pointer. This allows the compiler to omit the null check during conversion.

#pragma param_never_null *param_name* [...]

This pragma must immediately precede a function definition. It specifies a name or a list of space-separated names, which must correspond to the parameter names declared in the function definition. It checks that the named parameter is a class pointer type. Using this information it will generate more efficient code for a conversion from a pointer to a derived class to a pointer to a base class. It removes the need to check for the null pointer during the conversion.

For example,

```
#include <iostream>
using namespace std;
class A {
    int a;
};
class B {
    int b;
};
class C: public A, public B {
    int c;
};

C obj;
B *bpart = &obj;
bool fail = false;

#pragma param_never_null pc
void func(C *pc)
{
    B *pb;
    pb = pc;    /* without pragma the code generated has to
                  check for NULL */
    if (pb != bpart)
        fail = true;
}

int main(void)
{
    func(&obj);
    if (fail)
        cout << "Test failed" << endl;
    else
        cout << "Test passed" << endl;
    return 0;
}
```

#pragma suppress_null_check

This pragma must immediately precede an assignment of two pointers or a declaration list.

If the pragma precedes an assignment it indicates that the second operand pointer is not null and generates more efficient code for a conversion from a pointer to a derived class to a pointer to a base class. It removes the need to check for the null pointer before assignment.

On a declaration list it marks all variables as not being the null pointer. If the declaration contains an initialization expression, that expression is not checked for null.

For example,

```
#include <iostream>
using namespace std;
class A {
    int a;
};
class B {
    int b;
};
class C: public A, public B {
    int c;
};

C obj;
B *bpart = &obj;
bool fail = false;

void func(C *pc)
{
    B *pb;
    #pragma suppress_null_check
    pb = pc;    /* without pragma the code generated has to
                  check for NULL */
    if (pb != bpart)
```

```

        fail = true;
    }

void func2(C *pc)
{
    #pragma suppress_null_check
    B *pb = pc, *pb2 = pc; /* pragma means these initializations
                           need not check for NULL. It also marks pb and
pb2
                           as never being NULL, so the compiler will not
                           generate NULL checks in class conversions using
                           these pointers. */
    if (pb != bpart || pb2 != bpart)
        fail = true;
}

int main(void)
{
    func(&obj);
    func2(&obj);
    if (fail)
        cout << "Test failed" << endl;
    else
        cout << "Test passed" << endl;
    return 0;
}

```

Template Instantiation Pragmas

The **template instantiation pragmas** (`instantiate`, `do_not_instantiate` and `can_instantiate`) give fine-grain control over where (that is, in which object file) the individual instances of template functions, and member functions and static members of template classes are created. The creation of these instances from a template is called instantiation. As templates are a feature of C++, these pragmas are allowed only in `-c++` mode.

Refer to [“Compiler C++ Template Support” on page 1-344](#) for more information on how the compiler handles templates.

These pragmas take the name of an instance as a parameter, as shown in [Table 1-20](#).

Table 1-20. Instance Names

Name	Parameter
a template class name	A<int>
a template class declaration	class A<int>
a member function name	A<int>::f
a static data member name	A<int>::I
a static data declaration	int A<int>::I
a member function declaration	void A<int>::f(int, char)
a template function declaration	char* f(int, float)

If instantiation pragmas are not used, the compiler chooses object files in which to instantiate all required instances automatically during the pre-linking process.

#pragma instantiate *instance*

This pragma requests the compiler to instantiate *instance* in the current compilation.

For example,

```
#pragma instantiate class Stack<int>
```

causes all static members and member functions for the `int` instance of a template class `Stack` to be instantiated, whether they are required in this compilation or not. The example,

```
#pragma instantiate void Stack<int>::push(int)
```

causes only the individual member function `Stack<int>::push(int)` to be instantiated.

#pragma do_not_instantiate *instance*

This pragma directs the compiler not to instantiate *instance* in the current compilation. For example,

```
#pragma do_not_instantiate int Stack<float>::use_count
```

prevents the compiler from instantiating the static data member `Stack<float>::use_count` in the current compilation.

#pragma can_instantiate *instance*

This pragma tells the compiler that if *instance* is required anywhere in the program, it should be instantiated in this compilation.



Currently, this pragma forces the instantiation even if it is not required anywhere in the program. Therefore, it has the same effect as `#pragma instantiate`.

Header File Control Pragas

The header file control pragmas (`hdrstop`, `no_implicit_inclusion`, `no_pch`, `once`, and `system_header`) help the compiler to handle header files.

#pragma hdrstop

This pragma is used with the `-pch` (precompiled header) switch (on [page 1-55](#).) The switch tells the compiler to look for a precompiled header (`.pch` file), and, if it cannot find one, to generate a file for use on a later compilation. The `.pch` file contains a snapshot of all the code preceding the header stop point.

By default, the header stop point is the first non-preprocessing token in the primary source file. The `#pragma hdrstop` can be used to set the point earlier in the source file.

In the example,

```
#include "standard_defs.h"
#include "common_data.h"
#include "frequently_changing_data.h"

int i;
```

the default header stop point is start of the declaration of `i`. This might not be a good choice, as in this example, “`frequently_changing_data.h`” might change frequently, causing the `.pch` file to be regenerated often, and, therefore, losing the benefit of precompiled headers. The `hdrstop` pragma can be used to move the header stop to a more appropriate place.

For the following example,

```
#include "standard_defs.h"
#include "common_data.h"
#pragma hdrstop
#include "frequently_changing_data.h"

int i;
```

the precompiled header file would not include the contents of `frequently_changing_data.h`, as it is included after the `hdrstop` pragma, and so the precompiled header file would not need to be regenerated each time `frequently_changing_data.h` was modified.

#pragma no_implicit_inclusion

When the `-c++` switch ([on page 1-22](#)) is used for each included `.h` file, the compiler attempts to include the corresponding `.c` or `.cpp` file. This is called implicit inclusion.

If `#pragma no_implicit_inclusion` is placed in an `.h` file, the compiler does not implicitly include the corresponding `.c` or `.cpp` file with the `-c++` switch. This behavior only affects the `.h` file with `#pragma no_implicit_inclusion` within it and the corresponding `.c` or `.cpp` files.

For example, if there are the following files

```
t.c
```

which contains

```
#include "m.h"
```

and `m.h` and `m.c` are both empty, then

```
cc21k -c++ t.c -M
```

shows the following dependencies for `t.c`:

```
t.doj: t.c
```

```
t.doj: m.h
```

```
t.doj: m.c
```

If the following line is added to `m.h`,

```
#pragma no_implicit_inclusion
```

running the compiler as before would not show `m.c` in the dependencies list, such as:

```
t.doj: t.c
```

```
t.doj: m.h
```

#pragma no_pch

This pragma overrides the `-pch` (precompiled headers) switch (on [page 1-55](#)) for a particular source file. It directs the compiler not to look for a `.pch` file and not to generate one for the specified source file.

#pragma once

This pragma, which should appear at the beginning of a header file, tells the compiler that the header is written in such a way that including it several times has the same effect as including it once. For example,

```
#pragma once
#ifndef FILE_H
#define FILE_H
... contents of header file ...
#endif
```



In this example, the `#pragma once` is actually optional because the compiler recognizes the `#ifndef/#define/#endif` idiom and does not reopen a header that uses it.

#pragma system_header

This pragma identifies an `include` file as a file supplied with VisualDSP++. The VisualDSP++ compiler makes use of this information to help optimize uses of the supplied library functions and inline functions that these files define. The pragma should not be used in user application source.

Inline Control Pragmas

The compiler supports two pragmas to control the inlining of code. These pragmas are `#pragma always_inline`, `#pragma inline` and `#pragma never_inline`.

#pragma always_inline

This pragma may be applied to a function definition to indicate to the compiler that the function should always be inlined, and never called “out of line”. The pragma may only be applied to function definitions with the

`inline` qualifier, and may not be used on functions with variable-length argument lists. It is invalid for function definitions that have interrupt-related pragmas associated with them.

If the function in question has its address taken, the compiler cannot guarantee that all calls are inlined, so a warning is issued.

See [“Function Inlining” on page 1-108](#) for details of pragma precedence during inlining.

The following are examples of the `always_inline` pragma.

```
int func1(int a) {           // only consider inlining
    return a + 1;           // if -Oa switch is on
}

inline int func2(int b) {/// probably inlined, if optimizing
    return b + 2;
}

#pragma always_inline
inline int func3(int c) {/// always inline, even unoptimized
    return c + 3;
}

#pragma always_inline
int func4(int d) {/// error: not an inline function
    return d + 4;
}
```

#pragma inline

This pragma instructs the compiler to inline the function if it is considered desirable. The pragma is equivalent to specifying the `inline` keyword, but may be applied when the `inline` keyword is not allowed (such as when compiling in MISRA-C mode). [For more information, see “MISRA-C Compiler” on page 1-91.](#)

```
#pragma inline
int func5(int a, int b) { /* can be inlined */
    return a / b;
}
```

#pragma never_inline

This pragma may be applied to a function definition to indicate to the compiler that function should always be called “out of line”, and that the function’s body should never be inlined.

This pragma may not be used on function definitions that have the `inline` qualifier. See [“Function Inlining” on page 1-108](#) for details of pragma precedence during inlining.

These are code examples for the `never_inline` pragma.

```
#pragma never_inline
int func5(int e) { // never inlined, even with -Oa switch
    return e + 5;
}

#pragma never_inline
inline int func5(int f) { // error: inline function
    return f + 6;
}
```

Linking Control Pragmas

Linking pragmas (`linkage_name`, `core`, `section` and `weak_entry`) change how a given global function or variable is viewed during the linking stage.

#pragma linkage_name *identifier*

This pragma associates the identifier with the next external function declaration. It ensures that the identifier is used as the external reference, instead of following the compiler's usual conventions. For example,

```
_Pragma("linkage_name __realfuncname")  
void funcname ();
```

#pragma core

When building a project that targets multiple processors or multiple cores on a processor, a link stage may produce executables for more than one core or processor. The interprocedural analysis (IPA) framework requires that some conventions be adhered to in order to successfully perform its analyses for such projects.

Because the IPA framework collects information about the whole program, including information on references which may be to definitions outside the current translation unit, the IPA framework must be able to distinguish these definitions and their references without ambiguity.

If any confusion were allowed about which definition a reference refers to, then the IPA framework could potentially cause bad code to be generated, or could cause translation units in the project to be continually recompiled ad infinitum. It is the global symbols that are really relevant in this respect. The IPA framework will correctly handle locals and static symbols because multiple definitions are not possible within the same file, so there can be no ambiguity.

In order to disambiguate all references and the definitions to which they refer, it is necessary to have a unique name for each definition within a given project. It is illegal to define two different functions or variables with the same name. This is illegal in single-core projects because this would lead to multiple definitions of a symbol and the link would fail. In multi-core projects, however, it may be possible to link a project with multiple definitions because one definition could be linked into each link project, resulting in a valid link. Without detailed knowledge of what actions the linker had performed, however, the IPA framework would not be able to disambiguate such multiple definitions. For this reason, to use the IPA framework, it is up to you to ensure unique names even in projects targeting multiple cores or processors.

There are a few cases for which it is not possible to ensure unique names in multi-core or multi-processor projects. One such case is `main`. Each processor or core will have its own `main` function, and these need to be disambiguated for the IPA framework to be able to function correctly. Another case is where a library (or the C run-time startup) references a symbol which the user may wish to define differently for each core.

For this reason, VisualDSP++ 5.0 supports the `#pragma core(corename)`.

This pragma can be provided immediately prior to a definition or a declaration. This pragma allows you to give a unique identifier to each definition. It also allows you to indicate to which definition each reference refers. The IPA framework will use this core identifier to distinguish all instances of symbols with the same name and will therefore be able to carry out its analyses correctly.



Note that the `corename` specified should only consist of alphanumeric characters. Also note that the `corename` is case sensitive.

The pragma should be used:

- On every definition (not in a library) for which there needs to be a distinct definition for each core.
- On every declaration of a symbol (not in a library) for which the relevant definition includes the use of `#pragma core`. The core specified for a declaration must agree with the core specified for the definition.

It should be noted that the IPA framework will not need to be informed of any distinction if there are two identical copies of the same function or data with the same name. Functions or data that come from objects and that are duplicated in memory local to each core, for example, will not need to be distinguished. The IPA framework does not need to know exactly which instance each reference will get linked to because the information processed by the framework is identical for each copy. Essentially, the pragma only needs to be specified on items where there will be different functions or data with the same name incorporated into the executable for each core.

Here is an example of `#pragma core` usage to distinguish two different main functions:

```
/* foo.c */
#pragma core("coreA")
int main(void) {
    /* Code to be executed by core A */
}
/* bar.c */
#pragma core("coreB")
int main(void) {
    /* Code to be executed by core B */
}
```

Omitting either instance of the pragma will cause the IPA framework to issue a fatal error indicating that the pragma has been omitted on at least one definition.

Here is an example that will cause an error to be issued because the name contains a non-alphanumeric character:

```
#pragma core("core/A")
int main(void) {
    /* Code to executed on core A */
}
```

Here is an example where the pragma needs to be specified on a declaration as well as the definitions. There is a library which contains a reference to a symbol which is expected to be defined for each core. Two more modules define the `main` functions for the two cores. Two further modules, each only used by one of the cores, makes a reference to this symbol, and therefore requires use of the pragma. For example,

```
/* libc.c */
#include <stdio.h>
extern int core_number;
void print_core_number(void) {
    printf("Core %d\n", core_number);
}
/* maina.c */
extern void fooa(void)
#pragma core("coreA")
int core_number = 1;
#pragma core("coreA")
int main(void) {
    /* Code to be executed by core A */
    print_core_number();
    fooa();
}
/* mainb.c */
extern void foob(void)
#pragma core("coreB")
int core_number = 2;
#pragma core("coreB")
int main(void) {
    /* Code to be executed by core B */
    print_core_number();
}
```

```

        foob();
    }
    /* fooa.c */
    #include <stdio.h>
    #pragma core("coreA")
    extern int core_number;
    void fooa(void) {
        printf("Core: is core%c\n", 'A' - 1 + core_number);
    }
    /* foob.c */
    #include <stdio.h>
    #pragma core("coreB")
    extern int core_number;
    void foob(void) {
        printf("Core: is core%c\n", 'A' - 1 + core_number);
    }

```

In general, it will only be necessary to use `#pragma core` in this manner when there is a reference from outside the application (in a library, for example) where there is expected to be a distinct definition provided for each core, and where there are other modules that also require access to their respective definition. Notice also that the declaration of `core_number` in `lib.c` does not require use of the pragma because it is part of a translation unit to be included in a library.

A project that includes more than one definition of `main` will undergo some extra checking to catch problems that would otherwise occur in the IPA framework. For any non-template symbol that has more than one definition, the tool chain will fault any definitions that are outside libraries that do not specify a core name with the pragma. This check does not affect the normal behavior of the prelinker with respect to templates and in particular the resolution of multiple template instantiations.

To clarify:

Inside a library, `#pragma core` is not required on declarations or definitions of symbols that are defined more than once. However, a library can be responsible for forcing the application to define a symbol more than

once (that is, once for each core). In this case, the definitions and declarations require the pragma to be used outside the library to distinguish the multiple instances.

It should be noted that the tool chain cannot check that uses of `#pragma core` are consistent. If you use the pragma inconsistently or ambiguously then the IPA framework may end up causing incorrect code to be generated or causing continual recompilation of the application's files.

It is also important to note that the pragma does not change the linkage name of the symbol it is applied to in any way.

For more information on IPA, see [“Interprocedural Analysis” on page 1-84](#).

#pragma retain_name

This pragma indicates that the function or variable declaration that follows the pragma is not removed even though it apparently has no uses. Normally, when Interprocedural Analysis or linker elimination are enabled, the VisualDSP++ tools will identify unused functions and variables, and will eliminate them from the resulting executable to reduce memory requirements. The `retain_name` pragma instructs the tools to retain the specified symbol, regardless.

The following example shows how to use this pragma.

```
int delete_me(int x) {
    return x-2;
}

#pragma retain_name
int keep_me(int y) {
    return y+2;
}

int main(void) {
```



```
    return 0;
}
```

Since the program has no uses for either `delete_me()` or `keep_me()`, the compiler removes `delete_me()`, but keeps `keep_me()` because of the pragma. You do not need to specify `retain_name` for `main()`.

The pragma is only valid for global symbols. It is not valid for the following kinds of symbols:

- Symbols with `static` storage class
- Function parameters
- Symbols with `auto` storage class (locals). These are allocated on the stack at run-time.
- Members/fields within `structs/unions/classes`
- Type declarations

For more information on IPA, see [“Interprocedural Analysis” on page 1-84](#).

#pragma section/#pragma default_section

The section pragmas provide greater control over the sections in which the compiler places symbols.

The `section(SECTSTRING [, QUALIFIER, ...])` pragma is used to override the target section for any global or static symbol immediately following it. The pragma allows greater control over section qualifiers compared to the `section` keyword.

The `default_section(SECTKIND [, SECTSTRING [, QUALIFIER, ...]])` pragma is used to override the default sections in which the compiler is placing its symbols.

C/C++ Compiler Language Extensions

The default sections fall into the categories listed under *SECTKIND*. Except for the *STI* category, this pragma remains in force for a section category until its next use with that particular category, or the end of the file. The *STI* is an exception, in that only one *STI default_section* can be specified and its scope is the entire file scope, not just the part following the use of *STI*. A warning is issued if several *STI* sections are specified in the same file.

The omission of a section name results in the default section being reset to be the section that was in use at the start of the file, which can be either a compiler default value, or a value set by the user through the *-section* command line switch (for example, *-section SECTKIND=SECTSTRING*).

In all cases (including *STI*), the *default_section* pragma overwrites the value specified with the *-section* command line switch.

```
#pragma default_section(DATA, "NEW_DATA1")
int x;
#pragma default_section(DATA, "NEW_DATA2")
int x=5;
#pragma default_section(DATA, "NEW_DATA3")
int x;
```

In this case *x* is placed in *NEW_DATA2*, because the definition of *x* is within its scope.

A *default_section* pragma can only be used at global scope, where global variables are allowed.

SECTKIND can be one of the following keywords:

Table 1-21. *SECTKIND* Keywords

Keyword	Description
CODE	Section is used to contain procedures and functions
ALLDATA	Shorthand notation for DATA, CONSTDATA, BSS, STRINGS and AUTOINIT

Table 1-21. SECTKIND Keywords (Cont'd)

Keyword	Description
DATA	Section is used to contain “normal data”
CONSTDATA	Section is used to contain read-only data
BSZ	Section is used to contain uninitialized data
SWITCH	Section is used to contain jump-tables to implement C/C++ switch statements
VTABLE	Section is used to contain C++ virtual-function tables
STI	Section that contains code required to be executed by C++ initializations. For more information, see “Constructors and Destructors of Global Class Instances” on page 1-280.
STRINGS	Section that stores string literals
AUTOINIT	Contains data used to initialise aggregate autos.
PM_DATA	Section is used to contain normal data declared with <code>_pm</code> keyword
PM_CONSTDATA	Section is used to contain read-only data declared with <code>_pm</code> keyword

SECTSTRING is the double-quoted string containing the section name, exactly as it appears in the assembler file.

Changing one section kind has no effect on other section kinds. For instance, even though `STRINGS` and `CONSTDATA` are by default placed by the compiler in the same section, if `CONSTDATA default_section` is changed, the change has no effect on the `STRINGS` data.

Please note that ALLDATA is not a real section, but a rather pseudo-kind that stands for DATA, CONSTDATA, STRINGS, AUTOINIT and BSZ, and changing ALLDATA is equivalent to changing all of these section kinds. Therefore,

```
#pragma default_section(ALLDATA, params)
```

is equivalent with the sequence:

```
#pragma default_section(DATA, params)
#pragma default_section(CONSTDATA, params)
#pragma default_section(STRINGS, params)
#pragma default_section(AUTOINIT, params)
#pragma default_section(BSZ, params)
```

QUALIFIER can be one of the following keywords:

Table 1-22. QUALIFIER Keywords

Keyword	Description
PM	Section is located in program memory
DM	Section is located in data memory
ZERO_INIT	Section is zero-initialized at program startup
NO_INIT	Section is not initialized at program startup
RUNTIME_INIT	Section is user-initialized at program startup
DOUBLE32	Section may contain 32-bit but not 64-bit doubles
DOUBLE64	Section may contain 64-bit but not 32-bit doubles
DOUBLEANY	Section may contain either 32-bit or 64-bit doubles
SW	Code is short-word (2146x only).
NW	Code is normal-word (2146x only).

There may be any number of comma-separated section qualifiers within such pragmas, but they must not conflict with one another. Qualifiers must also be consistent across pragmas for identical section names, and

omission of qualifiers is not allowed even if at least one such qualifier has appeared in a previous pragma for the same section. If any qualifiers have not been specified for a particular section by the end of the translation unit, the compiler uses default qualifiers appropriate for the target processor. The following specifies that `f()` should be placed in a section "foo," which is `DOUBLEANY` qualified:

```
#pragma section("foo", DOUBLEANY)
void f() {}
```

The compiler always tries to honor the `section` pragma as its highest priority, and the `default_section` pragma is always the lowest priority.

For example, the following code results in function `f` being placed in section `foo`:

```
#pragma default_section(CODE, "bar")
#pragma section("foo")
void f() {}
```

The following code results in `x` being placed in section `zeromem`:

```
#pragma default_section(BSZ, "zeromem")
int x;
```

However, the following example does not result in the variable `a` being placed in section `onion` because it was declared with the `__pm` qualifier and therefore is placed in the PM data section:

```
#pragma default_section(DATA, "onion")
__pm int a = 4;
```

If the PM data is explicitly set as in this example,

```
#pragma default_section(PM_DATA, "pm_onion")
#pragma default_section(DATA, "onion")
__pm int a = 4;
```

then the variable `a` gets placed in the `pm_onion` section.


C/C++ Compiler Language Extensions

The following code results in code in section "foo" being compiled as short-word code (2146x processors only):

```
#pragma section("foo", SW)
```

The following results in code in section "foo2" being compiled as normal word code (2146x processors only):


```
#pragma default_section(CODE,"foo2", NW)
```

 In cases where a C++ STL object must be placed in a specific memory section, using `#pragma section/default_section` won't work. Instead, a non-default heap must be used, as explained in [“Allocating C++ STL Objects to a Non-Default Heap”](#).

#pragma file_attr("name[=value]" [, "name[=value]" [...]])

This pragma directs the compiler to emit the specified attributes when it compiles a file containing the pragma. Multiple `#pragma file_attr` directives are allowed in one file.

If “=value” is omitted, the default value of “1” will be used.

 The value of an attribute is all the characters after the '=' symbol and before the closing '"' symbol, including spaces. A warning will be emitted by the compiler if you have a preceding or trailing space as an attribute value, as this is likely to be a mistake.

See [“File Attributes” on page 1-348](#) for more information on using attributes.

#pragma weak_entry

This pragma may be used before a static variable or function declaration or definition. It applies to the function/variable declaration or definition that immediately follows the pragma. Use of this pragma causes the compiler to generate the function or variable definition with weak linkage.

The following are example uses of the `pragma weak_entry` directive.

```
#pragma weak_entry
int w_var = 0;
```

```
#pragma weak_entry
void w_func(){}
```



When a symbol definition is weak, it may be discarded by the linker in favor of another definition of the same symbol. Therefore, if any modules in the application make use of the `weak_entry` pragma, interprocedural analysis is disabled because it would be unsafe for the compiler to predict which definition will be selected by the linker. [For more information, see “Interprocedural Analysis” on page 1-84.](#)

Diagnostic Control Pragmas

The compiler supports `#pragma diag` which allows selective modification of the severity of compiler diagnostic messages.

The directive has three forms:

- Modify the severity of specific diagnostics
- Modify the behavior of an entire class of diagnostics
- Save or restore the current behavior of all diagnostics

Modifying the Severity of Specific Diagnostics

This form of the directive has the following syntax:

```
#pragma diag(ACTION: DIAG [, DIAG ...][: STRING])
```

The *action*: qualifier can be one of the following keywords:

Table 1-23. Keywords for Action Qualifier

Keyword	Action
suppress	Suppresses all instances of the diagnostic
dmaonly	Section is located in memory that can only be accessed by DMA. On ADSP-2126x and certain ADSP-2136x processors, this keyword applies to external memory.
remark	Changes the severity of the diagnostic to a remark.
warning	Changes the severity of the diagnostic to a warning.
error	Changes the severity of the diagnostic to an error.
restore	Restores the severity of the diagnostic to what it was originally at the start of compilation after all command-line options were processed.

If not in MISRA-C mode, the *diag* qualifier can be one or more comma-separated compiler diagnostic numbers without the preceding “cc” or zeros. The choice of error numbers is limited to those that may have their severity overridden (such as those that are displayed with a “{D}” in the error message). In addition, those diagnostics that are emitted by the compiler back-end (for example, after lexical analysis and parsing) cannot have their severity overridden either. Any attempt to override diagnostics that may not have their severity changed is silently ignored.

In MISRA-C mode, the *diag* qualifier is a list of MISRA-C rule numbers in the form *misra_rule_number_6_3* and *misra_rule_number_19_4* for rules 6.3 and 19.4, and so on. Special cases are rules 10.1 and 10.2, which are both split into four distinct rule checks. For example, 10.1(c) should be stated as *misra_rule_10_1_c*.

The third optional argument is a string-literal to insert a comment regarding the use of the `#pragma diag`.

Modifying the Behavior of an Entire Class of Diagnostics

This form of the directive has the following syntax, which is not allowed when in MISRA-C mode:

```
#pragma diag(ACTION)
```

The effects are as follows:

- `#pragma diag(errors)`
This pragma can be used to inhibit all subsequent warnings and remarks (equivalent to the `-w` switch option).
- `#pragma diag(remarks)`
This pragma can be used to enable all subsequent remarks and warnings (equivalent to the `-Wremarks` switch option).
- `#pragma diag(warnings)`
This pragma can be used to restore the default behavior when neither `-w` or `-Wremarks` is specified, which is to display warnings but inhibit remarks.

Saving or Restoring the Current Behavior of All Diagnostics

This form has the following syntax:

```
#pragma diag(ACTION)
```

The effects are as follows:

- `#pragma diag(push)`
This pragma may be used to store the current state of the severity of all diagnostic error messages.
- `#pragma diag(pop)`
This pragma restores all diagnostic error messages that was previously saved with the most recent `push`.

All `#pragma diag(push)` directives must be matched with the same number of `#pragma diag(pop)` directives in the overall translation unit, but need not be matched within individual source files, unless in MISRA-C mode. Note that the error threshold (set by the `remarks`, `warnings` or `errors` keywords) is also saved and restored with these directives.

The duration of such modifications to diagnostic severity are from the next line following the pragma to either the end of the translation unit, the next `#pragma diag(pop)` directive, or the next overriding `#pragma diag()` directive with the same error number. These pragmas may be used anywhere and are not affected by normal scoping rules.

All command-line overrides to diagnostic severity are processed first and any subsequent `#pragma diag()` directives will take precedence, with the restore action changing the severity back to that at the start of compilation after processing the command-line switch overrides.



Note that the directives to modify specific diagnostics are singular (for example, “error”), and the directives to modify classes of diagnostics are plural (for example, “errors”).

Memory Bank Pragma

The memory bank pragmas provide additional performance characteristics for the memory areas used to hold code and data for the function.

By default, the compiler assumes that there are no external costs associated with memory accesses. This strategy allows optimal performance when the code and data are placed into high-performance internal memory. In cases where the performance characteristics of memory are known in advance, the compiler can exploit this knowledge to improve the scheduling of generated code.

Note that memory banks are different from sections in the following ways:

- Section is a “hard” placement, using a name that is meaningful to the linker. If the `.ldf` file does not map the named section, a linker error occurs.
- A memory bank is a “soft” placement, using a name that is not visible to the linker. The compiler uses optimization to take advantage of the bank’s performance characteristics. However, if the `.ldf` file maps the code or data to memory that performs differently, the application still functions (albeit with a possible reduction in performance).

#pragma code_bank(*bankname*)

This pragma informs the compiler that the instructions for the immediately-following function are placed in a memory bank called *bankname*. Without this pragma, the compiler assumes that the instructions are placed into a bank called “`__code`”. When optimizing the function, the compiler takes note of attributes of memory bank *bankname*, and determines how long it takes to fetch each instruction from the memory bank.

In the example,

```
#pragma code_bank(slowmem)
int add_slowly(int x, int y) { return x + y; }
int add_quickly(int a, int b) { return a + b; }
```

the `add_slowly()` function is placed into the bank “`slowmem`”, which may have different performance characteristics from the “`__code`” bank, into which `add_quickly()` is placed.

#pragma data_bank(*bankname*)

This pragma informs the compiler that the immediately-following function uses the memory bank *bankname* as the model for memory accesses for non-local data that does not otherwise specify a memory bank. Without this pragma, the compiler assumes that non-local data should use the bank “__data” for behavioral characteristics.

In the example,

```
#pragma data_bank(green)
int green_func(void)
{
    extern int arr1[32];
    extern int bank("blue") i;
    i &= 31;
    return arr1[i++];
}
int blue_func(void)
{
    extern int arr2[32];
    extern int bank("blue") i;
    i &= 31;
    return arr2[i++];
}
```

In both `green_func()` and `blue_func()`, `i` is associated with the memory bank “blue”, and the retrieval and update of `i` are optimized to use the performance characteristics associated with memory bank “blue”.

The array `arr1` does not have an explicit memory bank in its declaration. Therefore, it is associated with the memory bank “green”, because `green_func()` has a specific default data bank. In contrast, `arr2` is associated with the memory bank “__data”, because `blue_func()` does not have a `#pragma data_bank` preceding it.

#pragma stack_bank(*bankname*)

This pragma informs the compiler that all locals for the immediately-following function are to be associated with memory bank *bankname*, unless they explicitly identify a different memory bank. Without this pragma, all locals are assumed to be associated with the memory bank “__stack”. In the example,

```
#pragma stack_bank(mystack)
short dotprod(int n, const short *x, const short *y)
{
    int sum = 0;
    int i = 0;
    for (i = 0; i < n; i++)
        sum += *x++ * *y++;
    return sum;
}
int fib(int n)
{
    int r;
    if (n < 2) {
        r = 1;
    } else {
        int a = fib(n-1);
        int b = fib(n-2);
        r = a + b;
    }
    return r;
}
#pragma interrupt
#pragma stack_bank(sysstack)
void count_ticks(void)
{
    extern int ticks;
    ticks++;
}
```

The `dotprod()` function places the `sum` and `i` values into the memory bank “`mystack`”, while `fib()` places `r`, `a` and `b` into the memory bank “`__stack`”, because there is no `stack_bank` pragma. The `count_ticks()` function does not declare any local data, but any compiler-generated local storage make use of the “`sysstack`” memory bank’s performance characteristics.

`#pragma bank_memory_kind(bankname, kind)`

This pragma informs the compiler what kind of memory the memory bank *bankname* is. The following kinds of memory are allowed by the compiler:

- Internal – the memory bank is high-speed in-core memory
- External – the memory bank is external to the processor

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

In the example,

```
#pragma bank_memory_kind(blue, internal)
int sum_list(bank("blue") const int *data, int n)
{
    int sum = 0;
    while (n--)
        sum += data[n];
    return sum;
}
```

the compiler knows that all accesses to the `data[]` array are to the “blue” memory bank, and hence to internal, in-core memory.

#pragma bank_read_cycles(*bankname*, *cycles*)

This pragma tells the compiler that each read operation on the memory bank *bankname* requires the *cycles* cycles before the resulting data is available. This allows the compiler to schedule sufficient code between the initiation of the read and the use of its results, to prevent unnecessary stalls.

In the example,

```
#pragma bank_read_cycles(slowmem, 20)
int dotprod(int n, const int *x, bank("slowmem") const int *y)
{
    int i, sum;
    for (i=sum=0; i < n; i++)
        sum += *x++ * *y++;
    return sum;
}
```

the compiler assumes that a read from **x* takes a single cycle, as this is the default read time, but that a read from **y* takes twenty cycles, because of the pragma.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition.

#pragma bank_write_cycles(*bankname*, *cycles*)

This pragma tells the compiler that each write operation on memory bank *bankname* requires the *cycles* cycles before it completes. This allows the compiler to schedule sufficient code between the initiation of the write and a subsequent read or write to the same location, to prevent unnecessary stalls.

In the following example,

```
void write_buf(int n, const char *buf)
{
    volatile bank("output") char *ptr = REG_ADDR;
```

C/C++ Compiler Language Extensions

```
        while (n--)\n            *ptr = *buf++;\n    }\n#pragma bank_write_cycles(output, 6)
```

the compiler knows that each write through `ptr` to the “output” memory bank takes six cycles to complete.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition. This is shown in the preceding example.

`#pragma bank_optimal_width(bankname, width)`

This pragma informs the compiler that *width* is the optimal number of bits to transfer to/from memory bank *bankname* in a single cycle. This can be used to indicate to the compiler that some memories can benefit from vectorization and similar strategies more than others. The *width* parameter must be 8, 16, 24 or 32.

In the example,

```
void memcpy_simple(char *dst, const char *src, size_t n)\n{\n    while (n--)\n        *dst++ = *src++;\n}\n#pragma bank_optimal_width(__code, 16)
```

the compiler knows that the instructions for the generated function would be best fetched in multiples of 16 bits, and so can select instructions accordingly.

The pragma must appear at global scope, outside any function definitions, but need not immediately precede a function definition. This is shown in the preceding example.

Code Generation Pragmas

The code generation pragmas are described below in the following sections:

#pragma avoid_anomaly_45 {on | off}

When executing code from external SDRAM on the ADSP-21161 processor, conditional instructions containing a DAG1 data access may not be performed correctly.

These pragmas, `#pragma avoid_anomaly_45 on` and `#pragma avoid_anomaly_45 off`, allow you to initiate (or avoid) the generation of such instructions on a function-by-function basis. The pragmas should be used before a function definition and remain in effect until another variant of the pragma is seen.

#pragma no_db_return

This pragma is used immediately before a function definition and will cause the compiler to ensure that non-delayed-branch instructions are used to return from the function. The pragma may be applied to both interrupt and non-interrupt function definitions. Applying the pragma to an interrupt function can be used as a workaround for ADSP-2137x silicon anomaly 09-00-0015, “Incorrect Popping of stacks possible when exiting IRQx/Timer interrupts with DB modifiers.”

If the pragma does not appear immediately before a function definition then a compiler error message is issued.

The following examples show uses of this pragma:

Example 1

```
#pragma no_db_return
int max(int x, int y)
{
    if (y > x)
```

```
        return y;
    else
        return x;
}
```

Example 2

```
#pragma no_db_return
#pragma interrupt_complete_nesting
void foo(void) {
    . . .
}
```

Example 3

```
#pragma no_db_return
int i; /* INVALID - not a function definition, causes compiler
error cc1943 */
```

Exceptions Table Pragma

The following is an exceptions table pragma.

#pragma generate_exceptions_tables

This pragma may be applied to a C function definition to request the compiler to generate tables which enable C++ exceptions to be thrown through executions of this function.

The following example consists of two source files. The first is a C file which contains the pragma applied to the definition of function `call_a_call_back`.

```
#pragma generate_exceptions_tables
void call_a_call_back(void pfn(void)) {
    pfn();/* without pragma program terminates when
        throw_an_int throws an exception */
}
```

The second source file contains C++ code. The function `main` calls `call_a_call_back`, from the C file listed above, which in turn calls `throw_an_int`. The exception thrown by `throw_an_int` will be caught by the catch handler in `main` because use of the `pragma` ensured the compiler generated an exceptions table for `call_a_call_back`.

```
#include <iostream>
extern "C" void call_a_call_back(void pfn());

static void throw_an_int() {
    throw 3;
}

int main() {
    try {
        call_a_call_back(throw_an_int);
    } catch (int i) {
        if (i == 3) std::cout << "Test passed\n";
    }
}
```

An alternative to using `#pragma generate_exceptions_tables` is to compile C files with the `-eh` (enable exception handling) switch (on page 1-32) which, for C files, is equivalent to using the `pragma` before every function definition.

GCC Compatibility Extensions

The compiler provides compatibility with the C dialect accepted by version 3.2 of the GNU C Compiler. Many of these features are available in the C99 ANSI Standard. A brief description of the extensions is included in this section. For more information, refer to the following web address:

<http://gcc.gnu.org/onlinedocs/gcc-3.2.1/gcc/C-Extensions.html#C%20Extensions>



The GCC compatibility extensions are only available in C dialect mode. They are not accepted in C++ dialect mode.

Statement Expressions

A statement expression is a compound statement enclosed in parentheses. A compound statement itself is enclosed in braces { }, so this construct is enclosed in parentheses-brace pairs ({ }).

The value computed by a statement expression is the value of the last statement which should be an expression statement. The statement expression may be used where expressions of its result type may be used. But they are not allowed in constant expressions.

Statement expressions are useful in the definition of macros as they allow the declaration of variables local to the macro. In the following example,

```
#define min(a,b) ({           \
    short __x=(a),__y=(b),__res;\
    if (__x > __y)           \
        __res = __y;        \
    else                     \
        __res = __x;        \
    __res;                   \
})

int use_min() {
    return min(foo(), thing()) + 2;
}
```

The `foo()` and `thing()` statements get called once each because they are assigned to the variables `__x` and `__y` which are local to the statement expression that `min` expands to and `min()` can be used freely within a larger expression because it expands to an expression.

Labels local to a statement expression can be declared with the `__label__` keyword. For example,

```
#define checker(p)    ({           \
    __label__ exit;    \
    int i;             \
    ...                \
    exit;              \
})
```

```

    for (i=0; p[i]; ++i) {
        int d = get(p[i]);
        if (!check(d)) goto exit;
        process(d);
    }
exit:
i;
}))

extern int g_p[100];
int checkit() {
    int local_i = checker(g_p);
    return local_i;
}

```



Statement expressions are not supported in C++ mode. Also, statement expressions are an extension to C originally implemented in the GCC compiler. Analog Devices support the extension primarily to aid porting code written for that compiler. When writing new code, consider using inline functions, which are compatible with ANSI/ISO standard C++ and C99, and are as efficient as macros when optimization is enabled.

Type Reference Support Keyword (typeof)

The `typeof(expression)` construct can be used as a name for the type of expression without actually knowing what that type is. It is useful for making source code that is interpreted more than once such as macros or include files more generic.

The `typeof` keyword may be used where ever a `typedef` name is permitted such as in declarations and in casts. For example,

```

#define abs(a) ({
    typeof(a) __a = a;
    if (__a < 0) __a = - __a;
})

```

```
    __a;                                \  
  })
```

shows `typeof` used in conjunction with a statement expression to define a “generic” macro with a local variable declaration.

The argument to `typeof` may also be a type name. Because `typeof` itself is a type name, it may be used in another `typeof(type-name)` construct. This can be used to restructure the C type declaration syntax.

For example,

```
#define pointer(T)    typeof(T *)  
#define array(T, N)  typeof(T [N])  
  
array (pointer (char), 4) y;
```

declares `y` to be an array of four pointers to `char`.



The `typeof` keyword is not supported in C++ mode.

The `typeof` keyword is an extension to C originally implemented in the GCC compiler. It should be used with caution because it is not compatible with other dialects of C/C++ and has not been adopted by the more recent C99 standard.

GCC Generalized Lvalues

A cast is an `lvalue` (may appear on the left hand side of an assignment) if its operand is an `lvalue`. This is an extension to C, provided for compatibility with GCC. It is not allowed in C++ mode.

A comma operator is an `lvalue` if its right operand is an `lvalue`. This is an extension to C, provided for compatibility with GCC. It is a standard feature of C++.

A conditional operator is an `lvalue` if its last two operands are `lvalues` of the same type. This is an extension to C, provided for compatibility with GCC. It is a standard feature of C++.

Conditional Expressions with Missing Operands

The middle operand of a conditional operator can be left out. If the condition is non-zero (true), then the condition itself is the result of the expression. This can be used for testing and substituting a different value when a pointer is NULL. The condition is only evaluated once; therefore, repeated side effects can be avoided. For example,

```
printf("name = %s\n", lookup(key)?:"-");
```

calls `lookup()` once, and substitutes the string “-” if it returns NULL. This is an extension to C, provided for compatibility with GCC. It is not allowed in C++ mode.

Hexadecimal Floating-Point Numbers

C99 style hexadecimal floating-point constants are accepted. They have the following syntax.

```
hexadecimal-floating-constant:
    {0x|0X} hex-significand binary-exponent-part [ floating-suffix ]
hex-significand: hex-digits [ . [ hex-digits ] ]
binary-exponent-part: {p|P} [+|-] decimal-digits
floating-suffix: { f | l | F | L }
```

The hex-significand is interpreted as a hexadecimal rational number. The digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of two by which the significand is to be scaled. The floating suffix has the same meaning that it has for decimal floating constants: a constant with no suffix is of type `double`, a constant with suffix `F` is of type `float`, and a constant with suffix `L` is of type `long double`.

Hexadecimal floating constants enable the programmer to specify the exact bit pattern required for a floating-point constant. For example, the declaration

```
float f = 0x1p-126f;
```

causes `f` to be initialized with the value `0x800000`.

Zero-Length Arrays

Arrays may be declared with zero length. This is an anachronism supported to provide compatibility with GCC. Use variable length array members instead.

Variable Argument Macros

The final parameter in a macro declaration may be followed by ... to indicate the parameter stands for a variable number of arguments.

For example,

```
#define trace(msg, args...)fprintf (stderr, msg, ## args);
```

can be used with differing numbers of arguments,

```
trace("got here\n");  
trace("i = %d\n", i);  
trace("x = %f, y = %f\n", x, y);
```

The ## operator has a special meaning when used in a macro definition before the parameter that expands the variable number of arguments: if the parameter expands to nothing, then it removes the preceding comma.



The variable argument macro syntax comes from GCC. It is not compatible with C99 variable argument macros and is not supported in C++ mode.

Line Breaks in String Literals

String literals may span many lines. The line breaks do not need to be escaped in any way. They are replaced by the character \n in the generated string. This extension is not supported in C++ mode. The extension is not compatible with many dialects of C, including ANSI/ISO C89 and C99. However, it is useful in `asm` statements, which are intrinsically non-portable.

Arithmetic on Pointers to Void and Pointers to Functions

Addition and subtraction is allowed on pointers to `void` and pointers to functions. The result is as if the operands had been cast to pointers to `char`. The `sizeof()` operator returns one for `void` and function types.

Cast to Union

A type cast can be used to create a value of a union type, by casting a value of one of the unions' member's types.

Ranges in Case Labels

A consecutive range of values can be specified in a single case by separating the first and last values of the range with

For example,

```
case 200 ... 300:
```

Declarations Mixed With Code

In C mode, the compiler accepts declarations placed in the middle of code. This allows the declaration of local variables to be placed at the point where they are required. Therefore, the declaration can be combined with initialization of the variable.

For example, in the following function

```
void func(Key k) {
    Node *p = list;
    while (p && p->key != k)
        p = p->next;
    if (!p)
        return;
    Data *d = p->data;
    while (*d)
        process(*d++);
}
```

the declaration of `d` is delayed until its initial value is available, so that no variable is uninitialized at any point in the function.

Escape Character Constant

The character escape `'\e'` may be used in character and string literals and maps to the ASCII Escape code, 27.

Alignment Inquiry Keyword (`__alignof__`)

The `__alignof__ (type-name)` construct evaluates to the alignment required for an object of a type. The `__alignof__ expression` construct can also be used to give the alignment required for an object of the *expression type*.

If *expression* is an *lvalue* (may appear on the left hand side of an assignment), the alignment returned takes into account alignment requested by pragmas and the default variable allocation rules.

Keyword for Specifying Names in Generated Assembler (`asm`)

The `asm` keyword can be used to direct the compiler to use a different name for a global variable or function. (See also “[#pragma linkage_name identifier](#)” on page 1-195.) For example,

```
int N asm("C11045");
```

tells the compiler to use the label `C11045` in the assembly code it generates wherever it needs to access the source level variable `N`. By default, the compiler would use the label `_N`.

The `asm` keyword can also be used in function declarations but not function definitions. However, a definition preceded by a declaration has the desired effect.

For example,

```
extern int f(int, int) asm("func");

int f(int a, int b) {
    . . .
}
```

Function, Variable and Type Attribute Keyword (__attribute__)

The `__attribute__` keyword can be used to specify attributes of functions, variables and types, as in these examples,

```
void func(void) __attribute__((section("fred")));
int a __attribute__((aligned (8)));
typedef struct {int a[4];} __attribute__((aligned (4))) Q;
```

The `__attribute__` keyword is supported, and therefore code, written for GCC, can be ported. All attributes accepted by GCC on `ix86` are accepted. The ones that are actually interpreted by the `cc21k` compiler are described in the sections of this manual describing the corresponding pragmas. (See [“Pragmas” on page 1-156](#).)

Unnamed struct/union Fields Within struct/unions

The compiler allows you to define a structure or union that contains, as fields, structures and unions without names. For example:

```
struct {
    int field1;
    union {
        int field2;
        int field3;
    };
    int field4;
} myvar;
```

This allows the user to access the members of the unnamed union as though they were members of the enclosing struct; for example, `myvar.field2`.

C++ Fractional Type Support

While in C++ mode, the `cc21k` compiler supports fractional (fixed-point) arithmetic that provides a way of computing with non-integral values within the confines of the fixed-point representation. Hardware support for the 32-bit fractional arithmetic is available on the ADSP-21xxx processors.

Fractional values are declared with the `fract` data type. Ensure that your program includes the `<fract>` header file. The `fract` data type is a C++ class that supports a set of standard arithmetic operators used in arithmetic expressions. Fractional values are represented as signed values in a range of $[-1 \dots 1)$ with a binary point immediately after the sign bit.

Other value ranges are obtained by scaling or shifting. In addition to the arithmetic, assignment, and shift operations, `fract` provides several type-conversion operations.

For more information about supported fractional arithmetic operators, see [“Fractional Arithmetic Operations”](#). For sample programs demonstrating the use of the `fract` type, see [Listing 1-4 on page 1-326](#).



The current release of the software does not provide for automatic scaling of fractional values.

Format of Fractional Literals

Fractional literals use the floating-point representation with an “r” suffix to distinguish them from floating-point literals, for example, `0.5r`. The `cc21k` compiler validates fractional literal values at run time to ensure they reside within the valid range of values.

Fractional literals are written with the “r” suffix to avoid certain precision loss. Literals without an “r” are of the type `double`, and are implicitly converted to `fract` as needed. After the conversion of a 32-bit `double` literal to a `fract` literal, the value of the latter may retain only 25 bits of precision compared with the full 31 bits for a fractional literal with the “r” suffix.

Conversions Involving Fractional Values

The following notes apply to type-conversion operations:

- Conversion between a fractional value and a floating value is supported. The conversion to the floating-point type may result in some precision loss.
- Conversion between a fractional value and an integer value is supported. The conversion is not recommended because the only common values are 0 and -1.
- Conversion between a fractional value and a long double value is supported via `float` and may result in some precision loss.

Fractional Arithmetic Operations

The following notes summarize information about fractional arithmetic operators supported by the `cc21k` compiler:

- Standard arithmetic operations on two `fract` items include addition, subtraction, and multiplication.
- Assignment operations include `+=`, `-=`, and `*=`.
- Shift operations include left and right shifts. A left shift is implemented as a logical shift and a right shift is an arithmetic shift. Shifting left by a negative amount is not recommended.
- Comparison operations are supported between two `fract` items.

- Mixed-mode arithmetic has a preference for `fract`. For more information about the mixed-mode arithmetic, see “[Mixed Mode Operations](#)”.
- Multiplication of a fractional and an integer produces an integer result or a fractional result. The program context determines which type of result is generated following the conversion algorithm of C++. When the compiler does not have enough context, it generates an ambiguous operator message, for example:
error: more than one operator "*" matches these operands:
...
You should cast the result of the multiply operation if the error occurs.

Mixed Mode Operations

Most operations supported for fractional values are supported for mixed fractional/float or fractional/double arithmetic expressions. At run time, a floating-point value is converted to a fractional value, and the operation is completed using fractional arithmetic.

The assignment operations, such as `+=`, are the exception to the rule. The logic of an assignment operation is defined by the type of a variable positioned on the left side of the expression.

Floating-point operations require an explicit cast of a fractional value to the desired floating type.

Saturated Arithmetic

The `cc21k` compiler supports saturated arithmetic for fractional data in the saturated arithmetic mode.

Whenever a calculation results in a bigger value than the `fract` data type represents, the result is truncated (wrapped around). An overflow flag is set to warn the program that the value has exceeded its limits. To prevent

the overflow and to get the result as the maximum representable value when processing signal data, use saturated arithmetic. Saturated arithmetic forces an overflow value to become the maximum representable value.

The run-time environment does not change the saturation mode of the processor during initialization. The default mode (typically no saturation) is set by the DSP hardware at reset. (Consult the hardware reference manual of an appropriate processor for the reset state.) The mode can be changed by using `set_saturate_mode()` and `reset_saturate_mode()` functions. Each arithmetic operator has its corresponding variant effected in the saturated mode.

For example, `add_sat`, `sub_sat`, `neg_sat`, and so on.

SIMD Support

The ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors support Single-Instruction, Multiple-Data (SIMD) operations, which, under certain conditions, double the computational rate over ADSP-2106x processors. It is important to gain some understanding of the processor architecture to take advantage of SIMD mode.

This section contains:

- [“Using SIMD Mode with Multichannel Data” on page 1-233](#)
- [“Using SIMD Mode with Single-Channel Data” on page 1-234](#)
- [“Restrictions to Using SIMD” on page 1-235](#)
- [“SIMD_for Pragma Syntax” on page 1-237](#)
- [“Compiler Constraints on Using SIMD C/C++” on page 1-238](#)
- [“Impact of Anomaly #40 on SIMD” on page 1-239](#)
- [“Performance When Using SIMD C/C++” on page 1-240](#)

The ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors include a second processing element that has its own register file and computational units. When the second element is active, the processor operates in the SIMD mode—each computation executes on both processing elements, and each element operates independently on different (“multiple”) data.

The SIMD mode is effective for performing exactly the same calculations simultaneously on two parallel sets of data. As a special case — which is what the compiler supports—programs can use the SIMD mechanism to perform a single task (such as summing a vector), by dividing it into two parts and doing both parts in parallel, simultaneously.

Also, there are situations where it is effective to perform two copies of a task simultaneously, such as summing two separate vectors.

SIMD processing is intimately linked with the memory model. In Single-Instruction, Single-Data (SISD) processing (ADSP-2106x compatible mode), the processor fetches single values from memory, performs single arithmetic operations, and stores single values back. In ADSP-2116x SIMD mode, each memory reference fetches a pair of values (from the designated address and the following address), one into each of the two compute blocks. Arithmetic instructions are done in pairs, and paired results are written back.

When compiling for the ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors, the `cc21k` compiler automatically generates SIMD code wherever possible. However, there are occasions when the `cc21k` compiler does not generate SIMD code because the compiler does not have enough information to be sure that it is safe to use SIMD. If such a situation occurs, the compiler generates a warning, specifying the reason the automatic generation of SIMD code was disabled. This situation occurs most often in functions, where the data to be processed is passed as parameters.

The primary causes of disabling automatic SIMD generation are a lack of alias information or a lack of alignment information. Normally, IPA helps to resolve these issues automatically. However, even with IPA, there may be times when the compiler cannot determine if it is safe to use SIMD code. If SIMD code is appropriate, then the `SIMD_for` pragma can be used to indicate this to the compiler.

Using SIMD Mode with Multichannel Data

When processing multichannel data in SIMD mode, the program essentially runs two copies of the algorithm simultaneously. Multichannel processing could be used for a whole program, for processing two modem channels, or for more local processes, such as calculating the sine of two values simultaneously.

Because there are two copies of the algorithm running, there must be two copies of the data as well. Due to the ADSP-2116x/2126x/2136x/2137x SIMD memory architecture, the data must be interleaved in memory. The data for one channel uses only even locations, while data for the other channel uses the corresponding odd locations. Because the processor implicitly doubles the memory references, arranging data in memory can be as simple as allocating twice as much space for all variables. Correct data arrangement in memory depends on the algorithm.

Such a program could increment loop indices by 2 instead of 1, as each fetch has consumed two words of memory. Data that is common to both could be loaded with a broadcast load or duplicated in memory.

The ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors can handle conditional execution at a single instruction level in SIMD mode and counted loops. Programs cannot do a conditional branch that depends on the result of a computation in SIMD mode because there would be two different results (one for each channel) and the branch must go one way or the other.



The compiler does not provide extended C/C++ support for SIMD mode and multichannel data.

Using SIMD Mode with Single-Channel Data

When processing single-channel data in SIMD mode, most of the program operates in SISD mode. At key places where the program loops over a collection of contiguous data elements, the program enters SIMD mode to perform computations on both processing elements.

For example, a program adds two vectors element by element. The normal SISD code picks up elements one at a time, evaluating

```
c[j] = a[j] + b[j]
```

for each j . Since each element is processed independently, the operation can as well be done in pairs in SIMD mode, by placing:

```
c[j] = a[j] + b[j]
```

in one processing element, and

```
c[j+1] = a[j+1] + b[j+1]
```

in the other element. Note that the loop index now increments by 2. This kind of processing is an effective use of the SIMD capability.

SIMD processing can also be used when one of the terms is a scalar. In that case, you should make sure that the same scalar value is loaded into both compute blocks, and the rest of the SIMD processing is the same.

A useful variant of the SIMD loop occurs in a form called a reduction. In such a loop, a vector is reduced to a scalar value by the action of the loop. For example, summing a vector or calculating the dot product of two vectors are areas where a program could use reduction. Again, SIMD processing lets the program process the vector on both processor elements at the same time (half in each).

Note that a reduction loop has to compute a single result. To use SIMD processing, the SISD algorithm would be transformed slightly. Two partial results are accumulated with all the even elements contributing to one result and the odds to the other. At the end of the loop, the program must combine the partial results into a final one. The reduction approach has two effects for which you must account:

- If the data is floating-point type, the results likely differ slightly due to floating-point round-off differences.
- The final combination of the partial results takes a little time that detracts from the SIMD performance gain.

In any of the single channel cases, if the array contains an odd number of elements, an extra step of processing is required after the SIMD region. Note that when the number is not known at compile time, the extra step is conditional on a run-time check.



The compiler provides extended C/C++ support for SIMD mode and single channel data.

Restrictions to Using SIMD

Be aware that there are various implications when a program is transformed to process single-channel data in SIMD mode.

- The data and the access pattern must be arranged for SIMD fetches, which are always at immediately adjoining locations. For example, a program that sums every third element of an array or the first column of a multi-dimensional array would not be a good candidate for SIMD, because the second fetch does not pick up the element for the next iteration.
- The data must always be aligned on double-word boundaries when in SIMD mode. The compiler attempts to align arrays properly in memory, so that operations on whole arrays work. Under certain circumstances, it is possible that some arrays which are placed in

named sections may be allocated on an odd word boundary, and may therefore be accessed incorrectly in SIMD mode. These circumstances are usually associated with use of the `RESOLVE` directive in an `.ldf` file or with data elimination by the linker. Therefore, it is suggested that the declaration of any data that is placed in a named section and could be accessed in SIMD mode is preceded by a `#pragma align 2` directive.

You must be careful about situations that force misalignment. This can occur by calling a function with an argument that points to an arbitrary location in an array. For instance, a function `func(A[j])`, where `func` expects a pointer or array parameter could have a data alignment problem if the value of `j` is odd. In this case, the parameter denotes a misaligned array, and an attempt to use SIMD processing within `func` fails.

Another SIMD failure situation arises when the reference pattern involves odd locations. A reference to `A[j-1]` fails. Similarly, programs cannot have locations that change between even and odd addresses (for example, `A[j+k]`). The FIR loop nest is an example of the latter.

Some ADSP-2116x, ADSP-2126x, ADSP-2136x, and ADSP-2137x processors have a 32-bit external bus. Due to the shorter bus width these processors do not allow for SIMD access to data placed in external memory. If data is placed in external memory, then the `-no-simd` switch (on page 1-47) or the `no_vectorization` pragma (on page 1-167) should be used to disable SIMD access to this data.



You have to be careful about any interaction or dependency between different iterations of the loop in SIMD. This is important because the SIMD processing changes the order of evaluation. The data for iteration `N+1` is actually fetched from memory before the results of iteration `N` are written back. Some programs get wrong answers if done in SIMD.

Looking at the example,

```
a[j] = a[j-1] + 2;
```

the current iteration uses the results from the previous one; if these results have not yet been written back, the current iteration is incorrect.

SIMD_for Pragma Syntax

The code transformation of a loop to run in SIMD mode involves one command. You indicate which loops are suitable for SIMD execution, and the compiler does the rest. Indicating that a loop should execute in SIMD mode takes the form of a pragma command

```
#pragma SIMD_for
```

which is placed ahead of the loop. As a preprocessing directive, this command must be alone on the line similar to a `#define` or `#include` command.

The compiler responds to this pragma by first checking whether the loop meets the SIMD guidelines. If compliant, the compiler transforms the loop so that the processing is done in SIMD mode. Among other things, the transformation involves changing the loop increment to 2, so that the vector elements are processed in SIMD pairs.

If the loop performs a reduction, partial results are calculated and combined at the end. Also, the compiler takes care of duplicating scalar values used within the loop. The following loop uses `#pragma SIMD_for`.

```
float sum, c, A[N];
...
sum = 0;
#pragma SIMD_for
    for (j=0; j<N; j++) {
        sum += c * A[j];
    }
```

The compiler transforms this loop as:

```
// declare SIMD temporaries
float t_sum[2], t_c[2];
// initialize both partial sums
t_sum[0] = t_sum[1] = 0;
// initialize both parts of scalar constant
t_c[0] = t_c[1] = c;
// ENTER_SIMD_MODE -- set machine mode
for (j=0; j<N; j+=2) {
    t_sum[0] += t_c[0] * A[j];
    // -- implicit SIMD processing performs:
    // t_sum[1] += t_c[1] * A[j+1];
}
// LEAVE_SIMD_MODE
// combine partial sums
sum = t_sum[0] + t_sum[1];
```

Compiler Constraints on Using SIMD C/C++

There are a number of conditions that limit when SIMD operations may occur. The compiler attempts to check these conditions and issues warnings or errors when it detects problems or possible problems.

The compiler usually avoids changing a program when the compiler is not certain that the transformed program produces the same results as the original.

The SIMD transformations are handled a bit differently for two reasons.

- You provide explicit direction to use SIMD. Therefore, the compiler assumes that you are aware of what is needed for SIMD operation and that you share responsibility for correct operation.
- Some of the SIMD constraints are difficult or impossible to verify at compile time. Therefore, the compiler is not fully conservative in checking, because if it were, few, if any loops would be accepted.

The compiler checks the conditions that can be checked. In many cases, the compiler can verify that a loop is unacceptable and rejects it for SIMD processing with a warning or error. Rejection occurs when there is an obvious dependency, obvious alignment problems, or a non-unit stride.

In other cases, the determining values are not available at compile time, and the compiler cannot be sure whether the loop can be safely transformed. In such cases, the compiler issues a warning and proceeds.

For some constraints—primarily the proper alignment of arrays that are parameters (arguments) of the function—the compiler assumes that conditions are acceptable and does not issue a warning.

There are two other restrictions on using `SIMD_for` loops:

- Function calls may not be made from within a `SIMD_for` loop.
- All data types used within a `SIMD_for` loop must have single-word base types. Long doubles, doubles in `double-size-64` mode, and structs should not to be used within a `SIMD_for` loop.

Impact of Anomaly #40 on SIMD

The SIMD read from internal memory with a Shadow Write FIFO hit does not always function correctly. This anomaly has been identified in the Shadow Write FIFOs that exist between the internal memory array of the ADSP-21160M processor and core / IOP buses that access the memory. (See *ADSP-21160 SHARC DSP Hardware Reference* for more details on shadow register operation.)

If performing SIMD reads which cross Long Word Address boundaries (for example, odd Normal Word addresses or non-Long Word boundary aligned Short Word addresses) and the data for the read is in the Shadow Write FIFO, the read results in revision 0.0 behavior for the read.

C/C++ Compiler Language Extensions

To avoid the anomaly #40, SIMD operations must always operate on double-word aligned vectors. To accomplish this type of operation, the compiler

- Allocates all static arrays on an appropriate double-word boundary
- Ensures that arrays on the stack are double-word aligned by ensuring the stack is aligned on an even word boundary.
- Generates SIMD operations when it knows it is operating with double-word-aligned elements. It is able to do this when arrays are defined statically or locally, or the arrays are arguments whose properties can be determined by Inter Procedural Analysis. A further requirement is that the initial index and increment are both explicit.

As a result of these precautions, SISD mode is used when array properties and indexing are not “visible” to the optimizer.

The compiler generates a warning when it fails to automatically generate a SIMD operation due to lack of alignment information. The user can then add the `#pragma SIMD_for` in such cases where they can be sure that alignment requirements are satisfied.

Performance When Using SIMD C/C++

When handling multichannel data in SIMD mode, the ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors can accomplish roughly twice as much useful work as ADSP-2106x processors. This is diluted slightly if data-dependent conditional blocks must be accommodated. The compiler does not support multichannel data operations in C/C++ code.

When handling single channel data in SIMD mode, C and C++ programs using single-channel SIMD portions usually show some performance improvement but fall short of the double-performance level for a variety of reasons.

In measuring the performance improvement in single channel SIMD, it is useful to isolate the SIMD portion and verify that it is performing as intended. The overall program speedup is often bounded by factors outside of the SIMD portion.

Any parallel-processing situation requires a small amount of overhead to coordinate the parallelism, and the ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors are no exception. Understanding this factor can help you evaluate where SIMD mode is likely to be most beneficial.

Specific items include:

- **Mode change:** The processor must switch into SIMD mode and back out. Each change takes two cycles.
- **Initialization of scalars:** Non-array values must be duplicated in order to have correct values for both processing elements. This takes a few instructions per value. This is a compiler restriction only—assembly programmers may be able to use the broadcast load facility.
- **Collection of partial results:** For reductions such as vector dot-product, the two partial results must be combined at the end. This typically requires a move and a final add or multiply, another 2 or 3 cycles.

All of these are fairly small items and have little effect provided that the size of the SIMD loop is large.

Note, also, that the size of the inner loop is halved when working in SIMD mode. Consider the following example, a filter with 40 coefficients.

- Inner loop before SIMD: 40 iterations
- Inner loop with SIMD: 20 iterations

Because the ADSP-21xxx processors can do a dot-product with a single instruction, the loop represents 20 cycles. If the SIMD overhead is 6 cycles, this operation on the ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors represents an overhead cost of 30%.

Actually, the true cost is a bit higher. Most loops require a little prologue code to achieve full processor efficiency. When the loop size is halved, the relative impact of the prologue—which remains a constant size—is increased. The prologue can lead to the loss of a few more percentage points off the performance.

Examples Using SIMD C

The following are two examples on how to use SIMD.

- [“Using SIMD C: Problem Cases—Data Increments”](#)
- [“Using SIMD C: Problem Cases—Data Alignment”](#)


Using SIMD C: Problem Cases—Data Increments

In SIMD mode, an assignment to or from a memory location refers to memory location `[A]` for the `PEx` processing element and memory location `[A+1]` for the `PEy` processing element. The `#pragma SIMD_for` takes advantage of these assignments by taking code containing a stride 1 loop, which addresses contiguous memory locations, and turning it into a stride 2 loop, addressing every second memory location.



In a potentially SIMD compatible loop, it is essential that the stride of a loop through an array is 1, so the compiler uses the correct memory locations. Any other value for the stride results in incorrect behavior.

The following matrix multiplication function demonstrates some stride issues.

 This code is NOT a valid use of the `SIMD_for` pragma.

```
float *matmul(void *x_input,
              void *y_input,
              void *output,
              int r,
              int s,
              int t)
{
    float *ipx, *ipy, *output_new;
    float tmp = 0;
    int i = 0, j = 0, k = 0;

    ipx = (float *) x_input;
    ipy = (float *) y_input;
    output_new = (float *) output;

    for (i = 0; i < r; i++) {
        for (k = 0; k < t; k++) {
            tmp = 0;
            #pragma SIMD_for
            for (j = 0; j < s; j++) {
                // The next line is wrong in SIMD mode
                tmp += ipx[j + (i * s)] * ipy[k + (j * t)];
            }
            output_new[k + (i * r)] = tmp;
        }
    }
    printf("SIMD\n");
    return output_new;
}
```

The line in **bold** text above reads from the memory location `ipy[k+(j*t)]`. The loop counter, `j`, is multiplied to calculate the offset into the array. In this example, the stride through the array can not be 1, rather it is `t`.

The SIMD and non-SIMD versions of this code address the following memory locations on each iteration of the innermost loop:

Table 1-24. SIMD and Non-SIMD Memory Locations

non-SIMD	SIMD
<code>ipy[k]</code>	<code>ipy[k], ipy[k+1]</code>
<code>ipy[k + t]</code>	<code>ipy[k+(2*t)], ipy[k+(2*t)+1]</code>
<code>ipy[k + (2*t)]</code>	<code>ipy[k+(4*t)], ipy[k+(4*t)+1]</code>
<code>ipy[k + (3*t)]</code>	<code>ipy[k+(6*t)], ipy[k+(6*t)+1]</code>
<code>ipy[k + (4*t)]</code>	<code>ipy[k+(8*t)], ipy[k+(8*t)+1]</code>

Each version addresses different memory locations, giving different results.

SIMD C Loop Counter Rules

- If the loop counter is multiplied within a loop to calculate an offset, do not use SIMD.
- If the inner loop counter is used to subscript a multi-dimensional array, it must only be used as the last subscript. Otherwise, do not use SIMD.

Using SIMD C: Problem Cases—Data Alignment

To work properly, SIMD calculations must only be used on arrays or other data that are double-word aligned. The compiler and libraries have some responsibility in ensuring that arrays meet this condition. All arrays, unions, and structures are guaranteed to be double-word aligned by the compiler, and functions such as `malloc()` only return double-word aligned memory.

There are some conditions in which you are responsible for determining whether the code and data are compatible with SIMD execution. This section describes some of the pitfalls of which you need to be aware.

Using Two-Dimensional Arrays

The following array,

```
int xyz[9][9];
```

would be double-word aligned by the compiler.

Each sub-array, however, would start at an offset of 9 from the previous array, so `xyz[1]`, `xyz[3]`, and subsequent elements would not be double word aligned. Trying to use these sub-arrays in SIMD mode creates problems. If the function `sum()` contains SIMD code, then the following is incorrect:

```
for (i = 0; i < 10; i++)
    total += sum( xyz[i] );
// leads to SIMD problems
```

SIMD C/C++ Data Alignment Rule

Two-dimensional arrays containing an odd number of rows or columns may lead to problems.

Adding To Array Offsets

The following is an example in which an offset is calculated using outer and inner loop counters. This code is NOT a valid use of the `SIMD_for` pragma.

```
for (k = 0; k < 20; ++k)
    #pragma SIMD_for
    for (i = 0; i < 20-k; ++i)
        output[i] += (input[i] + input[i+k]);
// leads to SIMD problems
```

In this case, every second iteration of the outer loop ($k=1$, $k=3$, etc.) results in incorrect code as the expression `input[i+k]` is a non-double-word aligned location. Note that this loop could be rewritten as:

```
for (k = 0; k < 20; ++k) {
    if (k % 2)
        for (i = 0; i < 20-k; ++i)
```

```
        output[i] += (input[i] + input[i+k]);
else
    #pragma SIMD_for
    for (i = 0; i < 20-k; ++i)
        output[i] += (input[i] + input[i+k]);
}
```

This SIMD version is only used when $k=0$, $k=2$, and subsequent even elements. This technique does not offer the full performance benefits of SIMD, but does offer about a 50% improvement.

Accessing External Memory on ADSP-2126X and 2136X Processors

On ADSP-2126x and some ADSP-2136x processors, it is not possible to access external memory directly from the processor core. The compiler provides some facilities to allow access to variables in external memory from C/C++ code, and to reduce the possibility of errors due to incorrect data placement.

Link-time Checking of Data Placement

Data which is placed in external memory on ADSP-2126x and 2136x processors must be defined using the `DMAONLY` qualifier of the `section` or `default_section` pragmas ([on page 1-201](#)). For example:

```
#pragma section("seg_extmem1", DMAONLY)
int extmem1[100];
```

The linker will perform additional checks to ensure that data marked as `DMAONLY` is not placed in internal memory, and that “normal” data is not placed in external memory. If data is placed incorrectly, the linker will issue an error.

Refer to the *VisualDSP++ 5.0 Linker and Utilities Manual* for additional information on LDF changes.

Inline Functions for External Memory Access

Two inline functions, `read_extmem` and `write_extmem`, are provided to transfer data between internal and external memory. A full description of these functions is provided in the *VisualDSP++ 5.0 Run-Time Library Manual for SHARC Processors*.

Support for Interrupts

The SHARC compiler and run-time libraries provide support for interrupts used by the SHARC processor. The supported interrupt dispatchers are listed below, along with important performance information, features and limitations. This section describes:

- [“Interrupt Dispatchers”](#)
- [“Interrupts and Circular Buffering” on page 1-251](#)
- [“Avoiding Self-Modifying Code” on page 1-251](#)
- [“Interrupt Nesting Restrictions on ADSP-2116x/2126x/2136x/2137x Processors” on page 1-252](#)
- [“Restriction on Use of Super-Fast Dispatcher on ADSP-2106x Processors” on page 1-252](#)

Interrupt Dispatchers

There are five types of the interrupt dispatcher, each providing different levels of functionality and performance. Each of the dispatchers is discussed in turn, starting with the slowest and most comprehensive. Note that for each dispatcher, two variants of the set-up functions are available: one which uses self-modifying code and one which does not. The non-self-modifying variants are discussed at the end of this section.

For the **circular buffer interrupt dispatcher**, use the `interruptcb()` or `signalcb()` functions to set up the interrupt. This dispatcher provides the following services:

- Saves all data registers, index registers, modify registers; saves all relevant Length registers and zeroes them before calling the ISR (Interrupt Service Routine); saves the volatile base registers. On platforms with a second processing element (ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors), the S registers are also saved.
- On ADSP-21020, ADSP-2106x, ADSP-2116x and ADSP-2126x processors, saves the contents of the loop counter stack, meaning that DO loops can be used safely in the ISR.
- Sends the interrupt number to the ISR as a parameter.
- On ADSP-2106x processors, requires approximately 176 cycles before calling the dispatcher and 106 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x/2137x processors, requires approximately 211 cycles before calling the dispatcher and 121 cycles to return to the interrupted code.
- Interrupt nesting is allowed.



Due to hardware restrictions on 213xx processors, the loop counter stack can not be saved and restored by this dispatcher.

For the **normal interrupt dispatcher**, use the `interrupt()` or `signal()` functions to set up the interrupt. This dispatcher provides the following services:

- Saves all data registers, index registers, modify registers; saves the volatile base registers. On platforms with a second processing element (ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors), the S registers are also saved.
- On ADSP-21020, ADSP-2106x, ADSP-2116x and ADSP-2126x processors, save the contents of the loop counter stack, meaning that DO loops can be used safely in the ISR.
- On ADSP-2106x processors, requires approximately 148 cycles before calling the dispatcher and 90 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x/2137x processors, requires approximately 183 cycles before calling the dispatcher and 109 cycles to return to the interrupted code.
- Sends the interrupt number type to the ISR as a parameter
- Interrupt nesting is allowed.



Due to hardware restrictions on 213xx processors, the loop counter stack can not be saved and restored by this dispatcher.

For the **fast interrupt dispatcher**, use the `interruptf()` or `signalf()` functions. This dispatcher provides the following services:

- Saves all scratch registers. Does not save the loop stack, therefore DO loop handling is restricted to 6 levels in total (specified in hardware). If the ISR uses one level of nesting, your code must not use more than five levels. The `-restrict-hardware-loops` switch (on [page 1-59](#)) controls the level of loop nesting that a function is using.
- Interrupt nesting is allowed.

- Does not send the interrupt number type to the ISR as a parameter.
- On ADSP-2106x processors, requires approximately 45 cycles before calling the dispatcher and 36 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x/2137x processors, requires approximately 40 cycles before calling the dispatcher and 26 cycles to return to the interrupted code.

For the **super-fast interrupt dispatcher**, use the `interrupts()` or `signals()` functions. This dispatcher provides the following services:

- Does not save the loop stack, therefore `DO` loop handling is restricted to six levels (specified in hardware). Interrupt nesting is disabled.
- Does not preserve changes to the `FLAGS` bits on the `ASTAT` register. Therefore, any changes to these bits by the interrupt handler will be lost when the interrupt is completed.
- Does not send the interrupt number type to the ISR as a parameter.
- Uses the alternate register set. As a result, interrupt nesting is disabled while the dispatcher and ISR are being executed.
- On ADSP-2106x processors, requires approximately 36 cycles before calling the dispatcher and 11 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x/2137x processors, requires approximately 34 cycles before calling the dispatcher and 10 cycles to return to the interrupted code.

The **pragma interrupt dispatcher** is intended for use with user-written assembly functions or C functions that have been compiled using `#pragma interrupt`. (See [“Interrupt Handler Pragmas” on page 1-162.](#)) Use the `interruptss()` or `signalss()` function to utilize this dispatcher. This dispatcher provides the following services:

- Relies on the compiler (or assembly routine) to save and restore all appropriate registers.
- Does not save the loop stack, therefore DO loop handling is restricted to six levels (specified in hardware).
- Does not send the interrupt number type to the ISR as a parameter.
- Interrupt nesting is allowed.
- On ADSP-2106x processors, requires approximately 29 cycles before calling the dispatcher and 24 cycles to return to the interrupted code; on ADSP-2116x/2126x/2136x/2137x processors, requires approximately 24 cycles before calling the dispatcher and 15 cycles to return to the interrupted code.

Interrupts and Circular Buffering

Only the circular buffer interrupt dispatcher and pragma interrupt dispatcher ensure that all `Length` registers are zeroed at the start of the interrupt handler. Since the compiler can generate circular buffer code automatically, you should ensure that you choose the correct dispatcher for your application. (Refer to [“Interrupt Dispatchers” on page 1-247.](#)) The `-no-circbuf` switch ([on page 1-44](#)) can be used to disable the automatic circular buffer code generation feature.

Avoiding Self-Modifying Code

The interrupt set-up functions (for example, `interruptf()`) use self-modifying code to set up the interrupts as this offers savings in execution time and code size. Non-self-modifying variants of all these functions are supplied and are suffixed with “`nsm`”. For example, to utilize the fast interrupt dispatcher, use the `interruptfnsm()` or `signalfnsm()` functions. The choice of a non-self-modifying function has no effect on the dispatcher used and no effect on the overall interrupt handling performance.

Interrupt Nesting Restrictions on ADSP-2116x/2126x/2136x/2137x Processors

For ADSP-2116x/2126x/2136x/2137x processors, the following restrictions exist:

On these platforms, the interrupt vector code explicitly saves the `ASTATx`, `ASTATy` and `MODE1` registers on the status stack using a `PUSH STS` instruction. For the timer, `VIRPT` and `IRQ0-2` interrupts, the save occurs in addition to the automatic save of these registers. This has the effect of reducing the maximum depth of nested interrupts to between 10 and 15 levels, depending on whether the timer, `VIRPT` and `IRQ0-2` interrupts are used.

Restriction on Use of Super-Fast Dispatcher on ADSP-2106x Processors

One of the interrupt dispatchers supplied with VisualDSP++, `interrupts()`, relies on the use of the alternate register set. Therefore, it is not possible to service another interrupt while the current interrupt is being serviced. To ensure this action, the interrupt enable bit (`IRPTEN`) is cleared by the `interrupts()` dispatcher while it is servicing an interrupt, and restored afterwards.

Under certain, unusual circumstances on the ADSP-2106x processors, the interrupt enable bit can be set while the `interrupts()` dispatcher is being executed. A nested interrupt can result, possibly causing data corruption or other problems in user code.

The problem occurs when a lower priority interrupt (LPI) occurs and then, immediately afterwards, a higher priority interrupt (HPI) occurs. The problem only appears if the LPI is being serviced using `interrupts()`. When the LPI occurs, the processor jumps to the appropriate point in the interrupt vector table and executes the relevant code. In the default vector table, the first instruction disables the `IRPTEN` register, stopping any further interrupts from being serviced. If, however, a HPI occurs

“immediately” after the LPI (before `IRPTEN` is disabled), the service routine of the higher priority is executed. There is a 1-cycle delay to allow the first instruction of the lower-priority service routine (the clearing of `IRPTEN`) to be executed. When the HPI completes, it sets `IRPTEN` and returns control to the LPI. The LPI executes, the `IRPTEN` bit is set, and nested interrupts are able to take place. Here is a brief summary of the sequence of events which can cause this problem:

1. Lower priority interrupt (LPI) occurs.
2. Higher priority interrupt (HPI) occurs.
3. First instruction of LPI is executed and clears `IRPTEN`.
4. HPI is executed:
 - Clear the `IRPTEN` register.
 - Execute handler and set the `IRPTEN` register.
5. LPI is executed:
 - The `IRPTEN` register was reset after HPI and is now set.
 - Problems may appear.



Note that this is a problem on ADSP-2106x processors only.

Restrictions on using Normal and Circular Buffer Interrupt Dispatchers on 2136x Processors

Hardware restrictions on 2136x processors mean that it is potentially unsafe to use the "circular buffer" interrupt dispatcher and "normal" interrupt dispatcher in applications that use certain arithmetic-based loops. The types of loops that are unsafe are:

- Single instruction arithmetic loops
- Arithmetic loops where the last but one instruction of the loop is a branching instruction (that is, a `CALL` or `JUMP` instruction)

C/C++ Compiler Language Extensions

- Arithmetic loops where the first instruction is a CALL instruction

An arithmetic loop take the following form:

```
DO end_label UNTIL EQ; // Or any other arithmetic condition

// Some code
end_label:
// Some code
```

The compiler supplied with VisualDSP++ will only generate counter-based loops, not arithmetic loops, and the VisualDSP++ run-time libraries have been checked to ensure they do not contain unsafe forms of arithmetic loops. The restrictions on the dispatchers therefore only apply to user-written assembly code or third-party code.

Migrating .ldf Files From Previous VisualDSP++ Installations

The .ldf files which have been used in VisualDSP++ 4.5 projects require updating before they can be used in VisualDSP++ 5.0.

The changes are described in:

- [“C++ Support Tables \(ctor, gdt\)” on page 1-254](#)
- [“ADSP-21375 Memory Map” on page 1-257](#)
- [“C++ Run-time Libraries Rationalization” on page 1-257](#)

Files for versions of VisualDSP++ prior to VisualDSP++ 4.5 will need to be updated according to the release notes for each intervening release.

C++ Support Tables (ctor, gdt)

Note that this change described below is *required*.

Linker changes in VisualDSP++ 5.0 make it possible for non-contiguous placement of highly-aligned data. This means that order of mapping in output memory sections is not necessarily maintained. This will result in linker warning `li2040` which can be avoided by using the `FORCE_CONTIGUITY` directive when contiguous placement is required, and `NO_FORCE_CONTIGUITY` otherwise.

The C++ static constructor mechanism (`seg_ctdm/seg_ctdm1`) and exceptions handling support (`.gdt/.gdt1`) use table inputs which are terminated using the sections ending in “1”. This requires contiguous placement of these sections, so use of `FORCE_CONTIGUITY` is recommended.

For example, replace:

```
#ifdef __cplusplus
    dxm_ctdm
    {
        __ctors = .; /* points to start of the section*/
        INPUT_SECTIONS( $OBJECTS(seg_ctdm) $LIBRARIES(seg_ctdm))
        INPUT_SECTIONS( $OBJECTS(seg_ctdm1)$LIBRARIES(seg_ctdm1))
    } > mem_ctdm
#endif

...
seg_dmda
{
    RESERVE(heaps_and_stack, heaps_and_stack_length = 32K,2)
    INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
#ifdef __cplusplus
    INPUT_SECTIONS( $OBJECTS(.gdt) $LIBRARIES(.gdt))
    INPUT_SECTIONS( $OBJECTS(.gdt1) $LIBRARIES(.gdt1))
    INPUT_SECTIONS( $OBJECTS(.frt) $LIBRARIES(.frt))
    INPUT_SECTIONS( $OBJECTS(.rtti) $LIBRARIES(.rtti))
    INPUT_SECTIONS( $OBJECTS(.cht) $LIBRARIES(.cht))
```

C/C++ Compiler Language Extensions

```
        INPUT_SECTIONS( $OBJECTS(.edt) $LIBRARIES(.edt))
        INPUT_SECTIONS( $OBJECTS(seg_vtbl) $LIBRARIES(seg_vtbl))
#endif

with:

#ifdef __cplusplus
    dx_e_ctdm
    {
        FORCE_CONTIGUITY
        __ctors = .; /* points to start of the section */
        INPUT_SECTIONS( $OBJECTS(seg_ctdm) $LIBRARIES(seg_ctdm))
        INPUT_SECTIONS( $OBJECTS(seg_ctdml)$LIBRARIES(seg_ctdml))
    } > mem_ctdm
#endif
...
#ifdef __cplusplus
    dx_e_gdt
    {
        FORCE_CONTIGUITY
        INPUT_SECTIONS( $OBJECTS(.gdt) $LIBRARIES(.gdt))
        INPUT_SECTIONS( $OBJECTS(.gdt1) $LIBRARIES(.gdt1))
    } > seg_dmda
#endif
    seg_dmda
    {
        RESERVE(heaps_and_stack, heaps_and_stack_length = 32K, 2)
        INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
#ifdef __cplusplus
        INPUT_SECTIONS( $OBJECTS(.frt) $LIBRARIES(.frt))
        INPUT_SECTIONS( $OBJECTS(.rtti) $LIBRARIES(.rtti))
        INPUT_SECTIONS( $OBJECTS(.cht) $LIBRARIES(.cht))
        INPUT_SECTIONS( $OBJECTS(.edt) $LIBRARIES(.edt))
        INPUT_SECTIONS( $OBJECTS(seg_vtbl) $LIBRARIES(seg_vtbl))
#endif
    }
#endif
```

For more information, see “Constructors and Destructors of Global Class Instances” on page 1-280.

ADSP-21375 Memory Map

Note that this change is *required* for any ADSP-21375 project.

In VisualDSP++ 4.5, the tools used an incorrect memory map for the ADSP-21375 processor. Affected sections are:

- `seg_pmco`
- `seg_pmda`
- `seg_dmda`
- `seg_stak`
- `seg_flash`

Refer to the MEMORY declaration of the `213xx/ldf/ADSP-21375.ldf` in your VisualDSP++ installation for the correct memory ranges.

C++ Run-time Libraries Rationalization

Note that the following change is optional.

In previous versions of VisualDSP++, it was necessary to link against `libcpp*.dlb`, `libcppprt*.dlb` and `libx*.dlb` when C++ exceptions support was required. In VisualDSP++ 5.0, it is only necessary to link against the `libcpp*.dlb` library. Therefore, it is possible to simplify your `.ldf` file by removing references to `libx*.dlb` and `libcppprt*.dlb` libraries.

Preprocessor Features

The compiler includes a preprocessor that lets you use preprocessor commands within your C/C++ source. [Table 1-25](#) lists these commands and provides a brief description of each. The preprocessor automatically runs before the compiler.

Table 1-25. Preprocessor Commands

Command	Description
<code>#define</code>	Defines a macro or constant.
<code>#elif</code>	Sub-divides an <code>#if ... #endif</code> pair.
<code>#else</code>	Identifies alternative instructions within an <code>#if ... #endif</code> pair.
<code>#endif</code>	Ends an <code>#if ... #endif</code> pair.
<code>#error</code>	Reports an error message.
<code>#if</code>	Begins an <code>#if ... #endif</code> pair.
<code>#ifdef</code>	Begins an <code>#ifdef ... #endif</code> pair and tests if macro is defined.
<code>#ifndef</code>	Begins an <code>#ifndef ... #endif</code> pair and tests if macro is not defined.
<code>#include</code>	Includes source code from another file.
<code>#line</code>	Outputs specified line number before preprocessing.
<code>#undef</code>	Removes macro definition.
<code>#warning</code>	Reports a warning message.
<code>#</code>	Converts a macro argument into a string constant.
<code>##</code>	Concatenates two strings.

Preprocessor commands are also useful for modifying the compilation. Using the `#include` command, you can include header files (`.h`) that contain code and/or data. A macro, which you declare with the `#define` preprocessor command, can specify simple text substitutions or complex

substitutions with parameters. The preprocessor replaces each occurrence of the macro reference found throughout the program with the specified value.

The preprocessor is separate from the compiler and has some features that may not be used within your C/C++ source file. For more information, see the *VisualDSP++ 5.0 Assembler and Preprocessor Manual*.

Predefined Preprocessor Macros

[Table 1-26](#) describes the predefined preprocessor macros.

Table 1-26. Predefined Preprocessor Macro Listing

Macro	Function
<code>__2106x__</code>	When compiling for the ADSP-21060, ADSP-21061, ADSP-21062, or the ADSP-21065L processors, <code>cc21k</code> defines <code>__ADSP2106x__</code> as 1.
<code>__2116x__</code>	When compiling for the ADSP-21160 or ADSP-21161 processors, <code>cc21k</code> defines <code>__2116x__</code> as 1.
<code>__2126x__</code>	When compiling for the ADSP-21261, ADSP-21262, ADSP-21266 or ADSP-21267 processors, <code>cc21k</code> defines <code>__2126x__</code> as 1.
<code>__2136x__</code>	When compiling for the ADSP-21362, ADSP-21363, ADSP-21364, ADSP-21365, ADSP-21366, ADSP-21367, ADSP-21368, or ADSP-21369 processors, <code>cc21k</code> defines <code>__2136x__</code> and <code>__213xx__</code> as 1.
<code>__2137x__</code>	When compiling for the ADSP-21371 and ADSP-21375 processors, <code>cc21k</code> defines <code>__2137x__</code> and <code>__213xx__</code> as 1.
<code>__2146x__</code>	When compiling for the ADSP-21462, ADSP-21465, ADSP-21467, or ADSP-21469 processors, <code>cc21k</code> defines <code>__2146x__</code> as 1.
<code>__214xx__</code>	When compiling for the ADSP-21462, ADSP-21465, ADSP-21467, or ADSP-21469 processors, <code>cc21k</code> defines <code>__214xx__</code> as 1.

Table 1-26. Predefined Preprocessor Macro Listing

Macro	Function
__ADSP21000__	cc21k always defines __ADSP21000__ as 1.
__ADSP21020__	cc21k defines __ADSP21020__ as 1 when you compile with the -proc ADSP-21020 command-line switch.
__ADSP21060__	cc21k defines __ADSP21060__ as 1 when you compile with the -proc ADSP-21060 command-line switch.
__ADSP21061__	cc21k defines __ADSP21061__ as 1 when you compile with the -proc ADSP-21061 command-line switch.
__ADSP21062__	cc21k defines __ADSP21062__ as 1 when you compile with the -proc ADSP-21062 command-line switch.
__ADSP21065L__	cc21k defines __ADSP21065L__ as 1 when you compile with the -proc ADSP-21065L command-line switch. When compiling for ADSP-2106x processors, two additional macros are defined as 1: __ADSP21000__ and __2106x__.
__ADSP21160__	cc21k defines __ADSP21160__ as 1 when you compile with the -proc ADSP-21160 command-line switch.
__ADSP21161__	cc21k defines __ADSP21161__ as 1 when you compile with the -proc ADSP-21161 command-line switch. When compiling for ADSP-2116x processors, three additional macros are defined as 1: __ADSP21000__, __2116x__ and __SIMDSHARC__.
__ADSP21261__	cc21k defines __ADSP21261__ as 1 when you compile with the -proc ADSP-21261 command-line switch.
__ADSP21262__	cc21k defines __ADSP21262__ as 1 when you compile with the -proc ADSP-21262 command-line switch.
__ADSP21266__	cc21k defines __ADSP21266__ as 1 when you compile with the -proc ADSP-21266 command-line switch.
__ADSP21267__	cc21k defines __ADSP21267__ as 1 when you compile with the -proc ADSP-21267 command-line switch. When compiling for ADSP-2126x processors, three additional macros are defined as 1: __ADSP21000__, __2126x__ and __SIMDSHARC__.

Table 1-26. Predefined Preprocessor Macro Listing

Macro	Function
<code>__ADSP21362__</code>	cc21k defines <code>__ADSP21362__</code> as 1 when you compile with the <code>-proc ADSP-21362</code> command-line switch.
<code>__ADSP21363__</code>	cc21k defines <code>__ADSP21363__</code> as 1 when you compile with the <code>-proc ADSP-21363</code> command-line switch.
<code>__ADSP21364__</code>	cc21k defines <code>__ADSP21364__</code> as 1 when you compile with the <code>-proc ADSP-21364</code> command-line switch.
<code>__ADSP21365__</code>	cc21k defines <code>__ADSP21365__</code> as 1 when you compile with the <code>-proc ADSP-21365</code> command-line switch.
<code>__ADSP21366__</code>	cc21k defines <code>__ADSP21366__</code> as 1 when you compile with the <code>-proc ADSP-21366</code> command-line switch.
<code>__ADSP21367__</code>	cc21k defines <code>__ADSP21367__</code> as 1 when you compile with the <code>-proc ADSP-21367</code> command-line switch.
<code>__ADSP21368__</code>	cc21k defines <code>__ADSP21368__</code> as 1 when you compile with the <code>-proc ADSP-21368</code> command-line switch.
<code>__ADSP21369__</code>	cc21k defines <code>__ADSP21369__</code> as 1 when you compile with the <code>-proc ADSP-21369</code> command-line switch. When compiling for ADSP-2136x processors, four additional macros are defined as 1: <code>__ADSP21000__</code> , <code>__2136x__</code> , <code>__213xx__</code> and <code>__SIMDSHARC__</code> .
<code>__ADSP21371__</code>	cc21k defines <code>__ADSP21371__</code> as 1 when you compile with the <code>-proc ADSP-21371</code> command-line switch.
<code>__ADSP21375__</code>	cc21k defines <code>__ADSP21375__</code> as 1 when you compile with the <code>-proc ADSP-21375</code> command-line switch. When compiling for ADSP-2137x processors, four additional macros are defined as 1: <code>__ADSP21000__</code> , <code>__2137x__</code> , <code>__213xx__</code> and <code>__SIMDSHARC__</code> .
<code>__ADSP21462__</code>	cc21k defines <code>__ADSP21462__</code> as 1 when you compile with the <code>-proc ADSP-21462</code> command-line switch.
<code>__ADSP21465__</code>	cc21k defines <code>__ADSP21465__</code> as 1 when you compile with the <code>-proc ADSP-21465</code> command-line switch.
<code>__ADSP21467__</code>	cc21k defines <code>__ADSP21467__</code> as 1 when you compile with the <code>-proc ADSP-21467</code> command-line switch.

Table 1-26. Predefined Preprocessor Macro Listing

Macro	Function
<code>__ADSP21469__</code>	cc21k defines <code>__ADSP21469__</code> as 1 when you compile with the <code>-proc ADSP-21469</code> command-line switch. When compiling for ADSP-2146x processors, four additional macros are defined as 1: <code>__ADSP21000__</code> , <code>__2146x__</code> , <code>__214xx__</code> and <code>__SIMDSHARC__</code> .
<code>__ANALOG_EXTENSIONS__</code>	cc21k defines <code>__ANALOG_EXTENSIONS__</code> as 1.
<code>__cplusplus</code>	cc21k defines <code>__cplusplus</code> as 199711L when compiling in C++ mode.
<code>__DATE__</code>	The preprocessor expands this macro into the preprocessing date as a string constant. The date string constant takes the form <code>Mmm dd yyyy</code> . (ANSI standard).
<code>__DOUBLES_ARE_FLOATS__</code>	cc21k defines <code>__DOUBLES_ARE_FLOATS__</code> as 1 when the size of the <code>double</code> type is the same as the single-precision <code>float</code> type. When the <code>-double-size-64</code> compiler switch is used (on page 1-29), the macro is not defined.
<code>__ECC__</code>	cc21k always defines <code>__ECC__</code> as 1.
<code>__EDG__</code>	cc21k always defines <code>__EDG__</code> as 1. This signifies that an Edison Design Group front-end is being used.
<code>__EDG_VERSION__</code>	cc21k always defines <code>__EDG_VERSION__</code> as an integral value representing the version of the compiler's front-end.
<code>__EXCEPTIONS</code>	cc21k defines <code>__EXCEPTIONS</code> as 1 when C++ exception handling is enabled (using the <code>-eh</code> command-line switch on page 1-32).
<code>__FILE__</code>	The preprocessor expands this macro into the current input file name as a string constant. The string matches the name of the file specified on the compiler's command-line or in a preprocessor <code>#include</code> command (ANSI standard).
<code>_LANGUAGE_C</code>	cc21k always defines <code>_LANGUAGE_C</code> as 1.
<code>__LINE__</code>	The preprocessor expands this macro into the current input line number as a decimal integer constant (ANSI standard).
<code>_MISRA_RULES</code>	cc21k defines <code>_MISRA_RULES</code> as 1 when compiling in MISRA-C mode.

Table 1-26. Predefined Preprocessor Macro Listing

Macro	Function
<code>_NO_LONGLONG</code>	cc21k always defines <code>_NO_LONGLONG</code> as 1.
<code>__NO_BUILTIN</code>	cc21k defines <code>__NO_BUILTIN</code> as 1 when you compile with the <code>-no-builtin</code> command-line switch (on page 1-43).
<code>__NORMAL_WORD_CODE__</code>	When compiling for ADSP-2146x processors, cc21k defines <code>__NORMAL_WORD_CODE__</code> as 1, when compiling in normal-word mode.
<code>__RTTI</code>	cc21k defines <code>__RTTI</code> as 1 when C++ run-time type information is enabled (using the <code>-rtti</code> command-line switch on page 1-76).
<code>__SHORT_WORD_CODE__</code>	When compiling for ADSP-2146x processors, cc21k defines <code>__SHORT_WORD_CODE__</code> as 1, when compiling in short-word mode. This is the default when compiling for ADSP-2146x processors.
<code>__SIGNED_CHARS__</code>	cc21k defines <code>__SIGNED_CHARS__</code> as 1. The macro is defined by default,
<code>__SIMDSHARC__</code>	When compiling for ADSP-2116x, ADSP-2126x, ADSP-2136, ADSP-2137x, and ADSP-2146x processors, cc21k defines <code>__SIMDSHARC__</code> as 1. The <code>__SIMDSHARC__</code> define is used to identify processors that are capable of executing SIMD code.
<code>__STDC__</code>	cc21k always defines <code>__STDC__</code> as 1.
<code>__STDC_VERSION__</code>	cc21k always defines <code>__STD_VERSION__</code> as 199409L when compiling in C mode.
<code>__TIME__</code>	The preprocessor expands this macro into the preprocessing time as a string constant. The date string constant takes the form <code>hh:mm:ss</code> (ANSI standard).
<code>__VERSION__</code>	The preprocessor expands this macro into a string constant containing the current compiler version.

Table 1-26. Predefined Preprocessor Macro Listing

Macro	Function
<code>__VERSIONNUM__</code>	The preprocessor defines <code>__VERSIONNUM__</code> as a numeric variant of <code>__VERSION__</code> constructed from the version number of the compiler. Eight bits are used for each component in the version number and the most significant byte of the value represents the most significant version component. As an example, a compiler with version 7.1.0.0 defines <code>__VERSIONNUM__</code> as 0x07010000 and 7.1.1.10 would define <code>__VERSIONNUM__</code> to be 0x0701010A.
<code>__VISUALDSPVERSION__</code>	The preprocessor defines this macro to be an eight-digit hexadecimal representation of the VisualDSP++ release, in the form 0xMMmmuurr, where: <ul style="list-style-type: none">– MM is the major release number– mm is the minor release number– uu is the update number– rr is “00”, and reserved for future use For example, VisualDSP++5.0 Update 1 would be 0x05000100.
<code>__WORKAROUNDS_ENABLED</code>	cc21k defines this macro to be 1 if any hardware workarounds are implemented by the compiler. This macro is set if the <code>-si-revision</code> switch (on page 1-62) has a value other than “none” or if any specific workaround is selected by means of the <code>-workaround</code> compiler switch (on page 1-69).

Writing Macros

A macro is a name standing for a block of text that the preprocessor substitutes. Use the `#define` preprocessor command to create a macro definition. When the macro definition has arguments, the block of text the preprocessor substitutes can vary with each new set of arguments.

Compound Macros

Whenever possible, use inline functions rather than compound macros. If compound macros are necessary, define such macros to allow invocation like function calls. This will make your source code easier to read and maintain. If you want your macro to extend over more than oneline, you must escape the newlines with backslashes. If your macro contains a string literal and you are using the `-no-multiline` switch ([on page 1-46](#)), then you must escape the newline twice, once for the macro and once for the string.

The following two code segments define two versions of the macro

SKIP_SPACES:

```
/* SKIP_SPACES, regular macro */
#define SKIP_SPACES ((p), limit) { \
    char *lim = (limit); \
    while (p != lim) { \
        if (*(p)++ != ' ') { \
            (p)--; \
            break; \
        } \
    } \
}
```

```
/* SKIP_SPACES, enclosed macro */
#define SKIP_SPACES (p, limit) \
do { \
    char *lim = (limit); \
    while ((p) != lim) { \
        if (*(p)++ != ' ') { \
            (p)--; \
            break; \
        } \
    } \
} while (0)
```

Preprocessor Features

Enclosing the first definition within the `do {...} while (0)` pair changes the macro from expanding to a compound statement to expanding to a single statement. With the macro expanding to a compound statement, you sometimes need to omit the semicolon after the macro call in order to have a legal program. This leads to a need to remember whether a function or macro is being invoked for each call and whether the macro needs a trailing semicolon or not. With the `do {...} while (0)` construct, you can pretend that the macro is a function and always put the semicolon after it.

For example,

```
/* SKIP_SPACES, enclosed macro, ends without ';' */
if (*p != 0)
    SKIP_SPACES (p, lim);
else ...
```

This expands to:

```
if (*p != 0)
    do {
        ...
    } while (0); /* semicolon from SKIP_SPACES (...); */
else ...
```

Without the `do {...} while (0)` construct, the expansion would be:

```
if (*p != 0)
{
    ...
}
/* semicolon from SKIP_SPACES (...); */
else
```

C/C++ Run-Time Model and Environment

This section describes the conventions that you must follow as you write assembly code that can be linked with C/C++ code. The description of how C/C++ constructs appear in assembly language are also useful for low-level program analysis and debugging.

This section provides a full description of the ADSP-21xxx run-time model, including the layout of the stack, data access, and call/entry sequence.

This section describes:

- [“C/C++ Run-Time Environment”](#)
- [“Constructors and Destructors of Global Class Instances”](#)
- [“Support for argv/argc” on page 1-282](#)
- [“Using Multiple Heaps” on page 1-283](#)
- [“Compiler Registers” on page 1-293](#)

This model applies to the compiler-generated code. Assembly programmers are encouraged to maintain stack conventions.

C/C++ Run-Time Environment

The C/C++ run-time environment is a set of conventions that C and C++ programs follow to run on ADSP-21xxx processors. Assembly routines that you link to C/C++ routines must follow these conventions.

[Figure 1-2](#) shows an overview of the run-time environment issues that you must consider as you write assembly routines that link with C/C++ routines.

C/C++ Run-Time Model and Environment

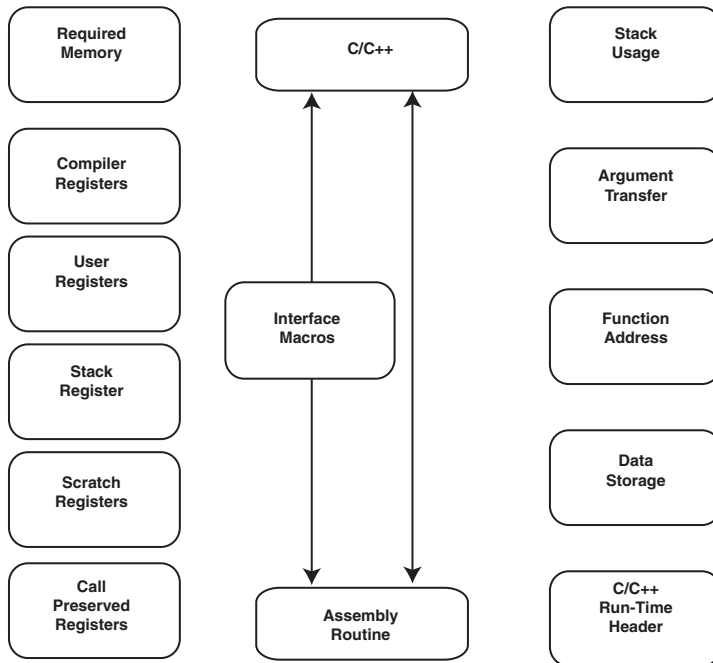


Figure 1-2. Assembly Language Interfacing Overview

These issues include:

- Register usage conventions:
 - “Compiler Registers” on page 1-293
 - “Miscellaneous Information About Registers” on page 1-294
 - “User Registers” on page 1-294
 - “Call Preserved Registers” on page 1-295
 - “Scratch Registers” on page 1-296
 - “Stack Registers” on page 1-297
- Memory usage conventions:
 - “Memory Usage” on page 1-270
 - “Memory Allocation for Stack and Heap on ADSP-2106x, 2116x and 2126x Processors” on page 1-277
 - “Measuring the Performance of the Compiler” on page 1-279
 - “Using Data Storage Formats” on page 1-307
- Program control conventions.
 - “Managing the Stack” on page 1-298
 - “Transferring Function Arguments and Return Value” on page 1-304
 - “Passing a C++ Class Instance” on page 1-306
 - “Using the Run-Time Header” on page 1-310

Memory Usage

The cc21k C/C++ run-time environment requires that a specific set of memory section names be used for placing code and data in memory. In assembly language files, these names are used as labels for the `.SECTION` directive. In the Linker Description File (`.ldf`), these names are used as labels for the output section names within the `SECTIONS{}` command. For information on how to control the sections used by the compiler, see [“#pragma section/#pragma default_section” on page 1-201](#)

For information on syntax for the Linker Description File and other information on the linker, see the *VisualDSP++ 5.0 Linker and Utilities Manual*. [Table 1-27](#) lists the memory section and output section names.

Because the compiler and linker must know the processor type to create code for the correct memory model, you must specify the processor for which you are developing. If you are using the VisualDSP++ IDDE, you specify the processor in the **Project Options** dialog box. If you are running the compiler from the command line, you specify the processor with a compiler switch. For more information on processor selection switches, see [“C/C++ Compiler Common Switch Descriptions” on page 1-23](#).

Table 1-27. Memory `.SECTION` and `SECTION{}` Names

Names	Usage Description
seg_pmco	This section must be in Program Memory (PM), holds code, and is required by some functions in the C/C++ run-time library. For more information, see “Program Memory Code Storage” on page 1-271 .
seg_swco	This section contains short-word instructions for targets that support VISA (variable instruction set) execution. It is used by the compiler and by functions in the short-word variants of the C/C++ run-time libraries. For more information, see “Program Memory Code Storage” on page 1-271 .
seg_dmda	This section must be in Data Memory (DM), is the default location for global and static variables and string literals, and is required by some functions in the C/C++ run-time library. For more information, see “Data Memory Data Storage” on page 1-272 .

Table 1-27. Memory .SECTION and SECTION{} Names (Cont'd)

Names	Usage Description
seg_pmda	This section must be in PM, holds PM data variables, and is required by some functions in the C/C++ run-time library. For more information, see “Program Memory Data Storage” on page 1-272 .
seg_stak	This section must be in DM, holds the run-time stack, and is required by the C/C++ run-time environment. For more information, see “Run-Time Stack Storage” on page 1-273 .
seg_heap	This section must be in DM, holds the default run-time heap, and is required by the C/C++ run-time environment. For more information, see “Run-Time Heap Storage” on page 1-273 .
seg_init	This section must be in PM, holds system initialization data, and is required for system initialization. For more information, see “Initialization Data Storage” on page 1-274 .
seg_rth	This section must be in the interrupt table area of PM, holds system initialization code and interrupt service routines, and is required for system initialization. For more information, see “Run-Time Header Storage” on page 1-275 .
seg_int_code	This section must always be located in internal memory. It contains library code that modifies the interrupt latch registers (IMASKP and IRPTL). A hardware anomaly on a number of SHARC processors means that it is unsafe for code located in external memory to modify these registers. This section is used to locate the affected library code in internal memory without restricting the location of the rest of the library code.
seg_int_code_sw	This section is for targets that support VISA execution. It contains short-word instructions that must be run from internal memory. It is used by some functions in the short-word variants of the C/C++ run-time libraries.

Program Memory Code Storage

For processors that do not support VISA execution, `seg_pmco` is the location where the compiler puts all the instructions that it generates when you compile your program. When linking, use your `.ldf` file to map this section to a Program Memory (PM) segment.

On processors that support VISA execution, the compiler puts all the instructions that it generates into `seg_swco` by default. When linking, use your `.ldf` file to map this section to an SW-qualified memory output section. When `-nwc` or `-normal-word-code` is used, the compiler puts all instructions into `seg_pmco`. When linking, use your `.ldf` file to map this section to a PM-qualified output section.

If you are assembling legacy assembly files and are using VISA execution support in your executable, use your `.ldf` file to map the input sections to an SW-qualified output section.

Data Memory Data Storage

The Data Memory data section, `seg_dmda`, is where the compiler puts global and static data and, for ADSP-210xx, 2116x and 2126x processors, the run-time stack and heap. When linking, use your `.ldf` file to map this section to DM space.

By default, the compiler stores static variables in the Data Memory data section. The compiler's `dm` and `pm` keywords (memory type qualifiers) let you override this default. If a memory type qualifier is not specified, the compiler places static and global variables in Data Memory. For more information on type qualifiers, see [“Dual Memory Support Keywords \(pm dm\)” on page 1-130](#). The following example allocates an array of 10 integers in the DM data section.

```
static int data [10];
```

Program Memory Data Storage

The Program Memory data section, `seg_pmda`, is where the compiler puts global and static data in Program Memory. When linking, use your `.ldf` file to map this section to PM space.

By default, the compiler stores static variables in the Data Memory data section. The compiler's `pm` keyword (memory type qualifier) lets you override this default and place variables in the Program Memory data section.

If a memory type qualifier is not specified, the compiler places static and global variables in Data Memory. For more information on type qualifiers, see [“Dual Memory Support Keywords \(pm dm\)” on page 1-130](#). The following example allocates an array of 10 integers in the PM data section.

```
static int pm coeffs[10];
```

Run-Time Stack Storage

On ADSP-2106x, 2116x and 2126x processors, the run-time stack memory is allocated from `seg_dmda`. On ADSP-213xx processors, the run-time stack is placed in `seg_heap`. Because the run-time environment cannot function without a stack, you must define one DM space. A typical size for the run-time stack is 4K 32-bit words of data memory.

The run-time stack is a 32-bit wide structure, growing from high memory to low memory. The compiler uses the run-time stack as the storage area for local variables and return addresses.

During a function call, the calling function pushes the return address onto the stack. (See [“Managing the Stack” on page 1-298](#).) For more information on configuring the run-time stack in the `.ldf` file, see [“Memory Allocation for Stack and Heap on ADSP-2106x, 2116x and 2126x Processors” on page 1-277](#).

Run-Time Heap Storage

On ADSP-2106x, 2116x and 2126x processors, the run-time heap memory is allocated from `seg_dmda`. On ADSP-213xx processors, the run-time stack is placed in `seg_stak`. A typical size for the run-time heap is 60K 32-bit words of data memory.

To dynamically allocate and deallocate memory at run-time, the C/C++ run-time library includes several functions: `malloc`, `calloc`, `realloc` and `free`. These functions allocate memory from the run-time heap by default.

The run-time library also provides support for multiple heaps, which allow dynamically allocated memory to be located in different blocks. See [“Using Multiple Heaps” on page 1-283](#) for more information on the use of multiple heaps. For more information on configuring the run-time heap in the `.ldf` file, see [“Memory Allocation for Stack and Heap on ADSP-2106x, 2116x and 2126x Processors” on page 1-277](#).



The Linker Description File requires the `seg_heap` declaration for every DSP project whether a program dynamically allocates memory at run-time or not.

Initialization Data Storage

The initialization section, `seg_init`, is where the compiler puts the initialization data in Program Memory. When linking, use your Linker Description File to map this section to Program Memory space.

The initialization section may be processed by two different utility programs: `mem21k` or `elfloader`.

- If you are producing boot-loadable executable file for your processor system, you should use the `elfloader` utility to process your executable. The `elfloader` utility processes your executable file, producing an ADSP-2106x boot-loadable file which you can use to boot a target hardware system and initialize its memory.

The boot loader, `elfloader`, operates on the executable file produced by the linker. When you run `elfloader` as part of the compilation process (using the `-no-mem` switch), the linker (by default) creates a `*.dxe` file for processing with `elfloader`.

When preparing files for the `elfloader` loader, the system configuration file's `seg_init` section needs only 16 slots/locations of space.

- If producing an executable file that is not going to be boot-loaded into the processor, you may use the `mem21k` utility to process your executable. The `mem21k` utility processes your executable file, pro-

ducing an optimized executable file in which all RAM memory initialization is stored in the `seg_init` PM ROM section. This optimization has the advantage of initializing all RAM to its proper value before the call to `main()` and reducing the size of an executable file by combining contiguous, identical initializations into a single block.

The memory initializer, `mem21k`, operates on the executable file produced by the linker. When running `mem21k` as part of the compilation process, the linker (by default) creates a `*.dxe` file for processing with `mem21k`.

The `mem21k` utility processes all the `PROGBITS` and `ZERO_INIT` sections except the initialization section (`seg_init`), the run-time header section (`seg_rth`), and the code section (`seg_pmco`). These sections contain the initialization routines and data.

The C run-time header reads the `seg_init` section, generated by `mem21k`, to determine which memory locations should be initialized to what values. This process occurs during the `__lib_setup_processor` routine that is called from the run-time header.

Run-Time Header Storage

The run-time header section, `seg_rth`, is where the compiler puts the system initialization code and interrupt table in Program Memory. When linking, use your `.ldf` file to map this section to the interrupt vector table area of Program Memory space.

If a run-time header file is not specified, the compiler uses a default run-time header from the appropriate `...lib` directory. [Table 1-28](#) lists these header files.

Note that if the compiler finds a copy of `xxx_hdr.obj` in the current directory, the compiler uses this copy instead of the file from the default directory.

Table 1-28. Header Files for Particular Targets

TARGET	HEADER FILE
21020	21k\lib\020_hdr.doj
21060	21k\lib\060_hdr.doj
21061	21k\lib\061_hdr.doj
21062	21k\lib\060_hdr.doj
21065L	21k\lib\065L_hdr.doj
21160	211xx\lib\160_hdr.doj
21161	211xx\lib\161_hdr.doj
21261	212xx\lib\261_hdr.doj
21262	212xx\lib\262_hdr.doj
21266	212xx\lib\266_hdr.doj
21267	212xx\lib\267_hdr.doj
21362	213xx\lib\362_hdr.doj
21363	213xx\lib\363_hdr.doj
21364	213xx\lib\364_hdr.doj
21365	213xx\lib\365_hdr.doj
21366	213xx\lib\366_hdr.doj
21367	213xx\lib\367_hdr.doj
21368	213xx\lib\368_hdr.doj
21369	213xx\lib\369_hdr.doj
21371	213xx\lib\371_hdr.doj
21375	213xx\lib\375_hdr.doj
21462	214xx\lib\21462_hdr.doj
21465	214xx\lib\21465_hdr.doj

Table 1-28. Header Files for Particular Targets (Cont'd)

TARGET	HEADER FILE
21467	214xx\lib\21467_hdr.doj
21469	214xx\lib\21469_hdr.doj

The source files for many run-time header files (including `060_hdr.asm` and `160_hdr.asm`) come with the development tools package. Keep the following points in mind if you prefer to write your own interrupt handlers in C/C++. Note that:

- The library functions `signal`, `raise`, `interrupt`, and their variants are based on the run-time header used.
- On the ADSP-21020 processor only, each interrupt is allocated eight words; on all other SHARC processors, each interrupt is allocated four words.

Memory Allocation for Stack and Heap on ADSP-2106x, 2116x and 2126x Processors

In previous releases of VisualDSP++, the default stack and heap were allocated separate memory sections in the LDFs. In VisualDSP++ 5.0, for ADSP-210xx, 2116x and 2126x processors, the allocation of memory for stacks and heaps is performed by the linker at link-time, resulting in more efficient memory use. (For ADSP-213xx processors, the stack and heap allocation remains the same because of the increased number of memory blocks.)

The memory for the stack and heap is allocated as follows:

- An area of memory in one of the default memory areas (for example, `seg_dmda`) is reserved for the stack and heap, using the `RESERVE()` command.
- Memory is allocated to data that must be placed in this section (for example, global variables and static variables).
- The `RESERVE_EXPAND()` command is used to claim any unused space in the default memory area and allocate it to the stack and heap. The ratio of memory allocated to the stack and heap can be adjusted if necessary.

Example of Heap/Stack Memory Allocation

[Listing 1-1 on page 1-278](#) shows how the `RESERVE()` command can be used to allocate memory for a heap and a stack in the `.ldf` file.

Listing 1-1. Heap/Stack Memory Allocation in LDFs

```
seg_dmda
{
    // Reserve a minimum of 32K for the stack and heap
    RESERVE(heap_and_stack, heap_and_stack_length = 32K)

    // Allocate space as necessary for libs and object files
    INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))

    // Expand the stack and heap space to fill the available
    // memory
    RESERVE_EXPAND(heap_and_stack, heap_and_stack_length)

    // Place the start and end markers for the stack. The
    // stack is allocated 25% (8K/32K) of the remaining space
    ldf_stack_space = heap_and_stack;

    ldf_stack_end = ldf_stack_space +
        ((heap_and_stack_length * 8K) / 32K);
}
```

```

ldf_stack_length = ldf_stack_end - ldf_stack_space;

// Place the start and end markers for the heap. The
// heap is allocated 75% (24K/32K) of the remaining space
ldf_heap_space = ldf_stack_end;
ldf_heap_end = ldf_heap_space +
               ((heaps_and_stack_length * 24K) / 32K);
ldf_heap_length = ldf_heap_end - ldf_heap_space;
} > seg_dmda

```



The following list contains the symbols that are used by the run-time libraries to create and manage the stack and heap. (These symbols must be defined in the `.ldf` file.)

<code>ldf_stack_space</code>	<code>ldf_stack_end</code>	<code>ldf_stack_length</code>
<code>ldf_heap_space</code>	<code>ldf_heap_end</code>	<code>ldf_heap_length</code>

Measuring the Performance of the Compiler

Benchmarking is done to measure the performance of a C compiler, or to understand how many processor clock cycles a specific section of code usually takes. Once the number of clock cycles is known, the amount of time that function takes to execute can be quickly calculated using the instruction rate of that processor. For more information, see “Measuring Cycle Counts” in Chapter 1 of the *VisualDSP++ *un-Time Library Manual for SHARC® Processors.**

Constructors and Destructors of Global Class Instances

Constructors for global class instances are invoked by the C/C++ run-time header during start-up. There are several components that allow this to happen:

- The associated data space for the instance
- The associated constructor (and destructor, if one exists) for the class
- A compiler-generated “start” routine
- A compiler-generated table of such “start” routines
- A compiler-constructed linked-list of destructor routines
- The run-time header itself

The interaction of these components is as follows.

The compiler generates a “start” routine for each module that contains globally-scoped class instances that need constructing or destructing. There is at most one “start” routine per module; it handles all the globally-scoped class instances in the modules:

- For each such instance, it invokes the instance's constructor. This may be a direct call, or it may be inlined by the compiler optimizer.
- If the instance requires destruction, the “start” routine registers this fact for later, by including pointers to the instance and its destructor into a linked list.

The start routine is named after the first such instance encountered, though the classes are not guaranteed to be constructed or destructed in any particular order (with the exception that destructors are called in the reverse order of the constructors). Such instances should not have any

dependency on construction order; the `-check-init-order` switch (on page 1-74) is useful for verifying this during system development, as it plants additional code to ensure objects are not constructed out of order.

A pointer to the “start” routine is placed into the `ctdm` section of the generated object file. When the application is linked, all `ctdm` sections are mapped into the same `ctdm` output section, forming a table of pointers to the “start” routines. An additional `ctdm1` object is appended to the end of the table; this contains a terminating NULL pointer.

When the run-time header is invoked, it calls `_ctor_loop()`, which walks the table of `ctdm` sections, calling each pointed-to “start” function until it reaches the NULL pointer from `ctdm1`. In this manner, the run-time header calls each global class instance’s constructor, indirectly through the pointers to “start” functions.

When the program reaches `exit()`, either by calling it directly or by returning from `main()`, the `exit()` routine follows the normal process of invoking the list of functions registered through the `atexit()` interface. One of these is a function that walks the list of destructors, invoking each in turn (in reverse order from the constructors).

This function is registers with `atexit()` during the run-time header, before `main()` is called.



Functions registered with `atexit()` may not make reference to global class instances, as the destructor for the instance may be invoked before the reference is used.

Constructors, Destructors and Memory Placement

By default, the compiler places the code for constructors and destructors into the same section as any other function’s code. This can be changed either by specifying the section specifically for the constructor or destructor (see “[#pragma section/#pragma default_section](#)” on page 1-201 and “[Placement Support Keyword \(section\)](#)” on page 1-136), or by altering

the default destination section for generated code (see “[#pragma section/#pragma default_section](#)” on page 1-201 and “[-section id=section_name\[,id=section_name...\]](#)” on page 1-60). Note that if a constructor is inlined into the “start” routine by the optimizer, such placement will have no effect. [For more information, see “Inlining and Sections” on page 1-112.](#)

While normal compiler-generated code is placed into the `CODE` area, the “start” routine is placed into the `STI` area. Both `CODE` and `STI` default to the same section, but may be changed separately using `#pragma default_section` or the `-section` switch (as the “start” function is an internal function generated by the compiler, its placement cannot be affected by `#pragma section`).

The pointer to the “start” routine is placed into the `ctdm` section. This is not configurable, as the invocation process relies on all of the “start” routine pointers being in the same section during linking, so that they form a table. It is essential that all relevant `ctdm` sections are mapped during linking; if a `ctdm` section is omitted, the associated constructor will not be invoked during start-up, and run-time behavior will be incorrect.

If destructors are required, the compiler generates data structures pointing to the class instance and destructor. These structures are placed into the default variable-data section (the `DATA` area).

Support for argv/argc

By default, the facility to specify arguments that are passed to your `main()` (`argv/argc`) at run-time is enabled. However, to correctly set up `argc` and `argv` requires additional configuration by the user. Modify your application as follows:

- Define your command-line arguments in C by defining a variable called “`__argv_string`”. When linked, your new definition overrides the default zero definition otherwise found in the C run-time library. For example,

```
extern const char __argv_string[] = "-in x.gif -out
y.jpeg";
```

The following optional step may also have to be performed:

- To use command-line arguments as part of profile-guided optimization (PGO), it is necessary to define `__argv_string` within a memory section called `MEM_ARGV`. Therefore, define a memory section called `MEM_ARGV` in your `.ldf` file and include the definition of `__argv_string` in it if you are using PGO. The default `.ldf` files do this for you if macro `IDDE_ARGS` is defined at link-time.

Using Multiple Heaps

The SHARC C/C++ run-time library supports the standard heap management functions `calloc`, `free`, `malloc`, and `realloc`. By default, these functions access the default heap, which is defined in the standard Linker Description File and the run-time header.

User written code can define any number of additional heaps, which can be located in any of the SHARC processor memory blocks. These additional heaps can be accessed either by the standard `calloc`, `free`, `malloc`, and `realloc` functions, or via the Analog Devices extensions `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`.

The primary use of alternate heaps is to allow dynamic memory allocation from more than one memory block. The ADSP-21xxx architecture allows two data accesses per cycle (in addition to a code access) if the memory locations are in different banks.

Declaring a Heap

Each heap must be declared with a `.VAR` directive in the `seg_init.asm` file and the `.ldf` file must declare memory and section placement for the heaps. The default `seg_init.asm` file declares one heap, `seg_heap`. The following customized `seg_init.asm` file shows how to declare two heaps: `seg_heap` and `seg_heaq`.

To use a custom `seg_init.asm`, assemble it and use it to replace the default `seg_init.doj` that is a member of the `libc.dlb` archive. For information on how to modify an archive file, see the *VisualDSP++ 5.0 Linker and Utilities Manual*.

For example,

```
#if defined(__SHORT_WORD_CODE__)
.section/nw seg_init;
#else
.section/pm seg_init;
#endif

/*
 * The following initializations rely on several values being
 * established externally, typically by the linker
 * description file.
 */

.extern ldf_stack_space;          /* The base of the stack */
.extern ldf_stack_length;         /* The length of the stack */
.extern ldf_heap_space;           /* The base of a primary DM heap
"seg_heap" */
.extern ldf_heap_length;          /* The length of heap "seg_heap"
*/
.extern ldf_heaq_space;           /* Base of a DM heap "seg_heaq" */
.extern ldf_heaq_length;          /* Length of heap "seg_heaq" */
```

```
/* The first two 48-bit words represent the heap name and the
heap location. The heap name must be exactly 8 characters long,
and the heap location should be either FFFFFFFF for DM or
00000001 for PM locations. The next 3 words set the heap's ini-
tialization value, size and length. The size and length are set
with macros whose values are calculated according the information
in the project's .ldf file. */
```

```
.__lib_heap_descriptions:
```

```
.global __lib_heap_space;
.var    __lib_heap_space[5] =
        0x7365675F6865, /* 'seg_he' */
        0x6170FFFFFFFF, /* 'ap'      */
        0,
        ldf_heap_space,
        ldf_heap_length;
```

```
/* Add more heap descriptions here */
```

```
.global __lib_heaq_space;
.var    __lib_heaq_space[5] =
        0x7365675F6865, /* 'seg_he' */
        0x6171FFFFFFFF, /* 'aq'      */
        0,
        ldf_heaq_space,
        ldf_heaq_length;
```

```
.global __lib_end_of_heap_descriptions;
.var    __lib_end_of_heap_descriptions = 0; /* Zero for end of list
*/
```

```
____lib_heap_descriptions.end:
```

As noted above, the calculation for a heap's size and length occur in the project's Linker Description File. When linking, the linker handles substitution of values to resolve the heap's definition (the `.VAR` directive in the `seg_init.asm` file).

[Listing 1-1 on page 1-278](#) shows how the `.ldf` file is used to define the symbols `ldf_heap_space` and `ldf_heap_length` for the default heap. The same mechanism would be used to define `ldf_heapq_space` and `ldf_heapq_length` for the additional heap.

Heap Identifiers

All heaps have two identifiers:

- Primary heap ID is the index of the descriptor for that heap in the heap descriptor table (in `seg_init.asm`). The primary heap ID of the default heap is always 0, and the primary IDs of user-defined heaps are set to 1, 2, 3, and so on.
- Each heap also has a unique 8-letter name associated with it. The heap ID can be obtained by calling the function `heap_lookup_name` with this name as its parameter. The name must be exactly eight characters long.

Allocating C++ STL Objects to a Non-Default Heap

C++ STL objects can be placed in a non-default heap through use of a custom allocator. To do this, you must first create your custom allocator. Below is an example custom allocator that you can use as a basis for your own. The most important part of `customalloc.h` in most cases is the `allocate` function, where memory is allocated to the STL object. Currently, the pertinent line of code assigns to the default heap (0):

```
Ty* ty = (Ty*) heap_malloc(0, n * sizeof(Ty));
```

By changing the first parameter of `heap_malloc()`, you can allocate to a different heap. Therefore:

- 0 would be the default heap
- 1, the first user heap
- 2, the second user heap

and so on.

Once you have created your custom allocator, you must inform your STL object to use it. Note that the standard definition for “list”:

```
list<int> a;
```

is the same as writing:

```
list<int, allocator<int> > a;
```

where “allocator” is the default allocator. Therefore, we can tell list “a” to use our custom allocator as follows:

```
list<int, customallocator<int> > a;
```

Once created, the list “a” can be used as normal. Also, `example.cpp` (below) is a simple example that shows the custom allocator being used.

customalloc.h

```
template <class Ty>
class customallocator {
public:
    typedef Ty value_type;
    typedef Ty* pointer;
    typedef Ty& reference;
    typedef const Ty* const_pointer;
    typedef const Ty& const_reference;
```

C/C++ Run-Time Model and Environment

```
typedef size_t size_type;
typedef ptrdiff_t difference_type;

template <class Other>
struct rebind { typedef customallocator<Other> other; };
pointer address(reference val) const { return &val; }
const_pointer address(const_reference val)
    const { return &val; }
customallocator(){}
customallocator(const customallocator<Ty>&){}
template <class Other>
customallocator(const customallocator<Other>&) {}
template <class Other>
customallocator<Ty>& operator=(const customallocator&)
    { return (*this); }

pointer allocate(size_type n, const void * = 0) {
    Ty* ty = (Ty*) heap_malloc(0, n * sizeof(Ty));
    cout << "Allocating 0x" << ty << endl;
    return ty;
}

void deallocate(void* p, size_type) {
    cout << "Deallocating 0x" << p << endl;
    if (p) free(p);
}

void construct(pointer p, const Ty& val)
    { new((void*)p)Ty(val); }
void destroy(pointer p) { p->~Ty(); }
size_type max_size() const { return size_t(-1); } };
```


example.cpp

```

#include <iostream>
#include <list>
#include <customalloc.h>    // include your custom allocator
using namespace std;
main(){
    cout << "creating list" << endl;
    list<int, customallocator<int> > a;
        // create list with custom allocator
    cout.setf(ios_base::hex, ios_base::basefield);
    cout << "pushing some items on the back" << endl;
    a.push_back(0xaaaaaaaa);    // push items as usual
    a.push_back(0xbbbbbbbb);
    while(!a.empty()){
        cout << "popping:0x" << a.front() << endl;
            //read item as usual
        a.pop_front();          //pop items as usual
    }
    cout << "finished." << endl;
}

```

Using Alternate Heaps with the Standard Interface

Alternate heaps can be accessed by the standard functions `calloc`, `free`, `malloc`, and `realloc`. The run-time library keeps track of a current heap, which initially is the default heap. The current heap can be changed any number of times at runtime by calling the function `set_alloc_type` with the new heap name as a parameter, or by calling `heap_switch` with the heap ID as a parameter.

The standard functions `calloc` and `malloc` always allocate a new object from the current heap. If `realloc` is called with a null pointer, it also allocates a new object from the current heap.

Previously allocated objects can be deallocated with `free` or `realloc`, or resized by `realloc`, even if the current heap is now different from when the object was originally allocated. When a previously allocated object is resized with `realloc`, the returned object is always in the same heap as the original object.



Multithreaded programs (using VDK) cannot use `set_alloc_type` or `heap_switch` to change the current heap from the default. Such programs can access alternate heaps through the alternate interface described in the next section.

Using the Alternate Heap Interface

The C run-time library provides the alternate heap interface functions `heap_calloc`, `heap_free`, `heap_malloc`, and `heap_realloc`. These routines work exactly the same as the corresponding standard functions without the “heap_” prefix, except that they take an additional argument that specifies the heap ID. These functions are completely independent of the current heap setting.

Objects allocated with the alternate interface functions can be freed with either the `free` or `heap_free` (or `realloc` or `heap_realloc`) functions. The `heap_free` function is a little faster than `free` since it does not have to search for the proper heap. However, it is essential that the `heap_free` or `heap_realloc` functions be called with the same heap ID that was used to allocate the object being freed. If it is called with the wrong heap ID, the object would not be freed or reallocated.

The actual entry point names for the alternate heap interface routines have an initial underscore; they are `_heap_calloc`, `_heap_free`, `_heap_lookup`, `_heap_malloc`, `_heap_realloc` and `_heap_switch`. The `stdlib.h` standard header file defines macro equivalents without the leading underscores.

C++ Run-Time Support for the Alternate Heap Interface

The C++ run-time library provides support for allocation and release of memory from an alternative heap via the new and delete operators.

Heaps should be initialized with the C run-time functions as described. These heaps can then be used via the new and delete mechanism by simply passing the heap ID to the new operator. There is no need to pass the heap ID to the delete operator as the information is not required when the memory is released.

The routines are used as in the example below.

```
#include <heapnew>

char *alloc_string(int size, int heapID)
{
    char *retVal = new(heapID) char[size];
    return retVal;
}

void free_string(char *aString)
{
    delete aString;
}
```

Example C Programs

The C programs below show how to allocate and initialize two types of heap interfaces.

Standard Heap Interface

```
// Example program using the standard heap interface
// Assumes that the user has created an additional heap,
// "seg_heaq", which is located in PM memory

#include <stdlib.h>
#include <stdio.h>
```

C/C++ Run-Time Model and Environment

```
void func(int * a, int pm * b);

main()
{
    int * x;
    int pm * y;
    int loop;

    x = malloc(1000);           // get 1K words of DM heap space
    set_alloc_type("seg_heap"); // Set the current heap to
"seg_heap"
    y = (int pm *)malloc(1000); // get 1K words of PM heap space
    set_alloc_type("seg_heap"); // Reset the current heap to
                                // "seg_heap" in case it is referred
                                // to elsewhere
    for (loop = 0; loop < 1000; loop++)
        x[loop] = y[loop] = loop;
    func(x, y);                // Do something with x and y
}
```

Alternate Heap Interface

```
// Example function using the alternate heap interface
// Assumes that the user has created an additional heap,
"seg_heap",
// which is located in PM memory
#include <stdlib.h>

void func(int * a, int pm * b);

main()
{
    int * x;
    int pm * y;
    int loop, pm_heapID;
    pm_heapID = heap_lookup_name("seg_heap");
    x = heap_malloc(0, 1000); // get 1K words of DM heap space
    y = (int pm *)heap_malloc(pm_heapID, 1000);
                                // get 1K words of PM heap space
}
```

```

    for (loop = 0; loop < 1000; loop++)
        x[loop] = y[loop] = loop;
    func(x, y);                // Do something with x and y
}

```

Compiler Registers

The cc21k C/C++ run-time environment reserves a set of registers for its own use. [Table 1-29](#) lists these registers and the values the C/C++ run-time environment expects to be in them. Do not modify these registers, except as noted in the table.

Table 1-29. Compiler Registers

Register	Value	Modification Rules
m5, m13	0	Do not modify
m6, m14,	1	Do not modify
m7, m15	-1	Do not modify
b6, b7	stack base	Do not modify
l6, l7	stack length	Do not modify
l0, l1, l2, l3, l4, l5, l8, l9, l10, l11, l12, l13, l14, l15	0	Modify for temporary use, restore when done
MMASK (ADSP-2116x/26x/36x processors only)	0xE03003	Do not modify if you are using the interrupt dispatchers supplied with VisualDSP++.

Miscellaneous Information About Registers

The following is some miscellaneous information that C/C++ and assembly programmers might find helpful in understanding register functionality:

- All of the L registers, except L6 and L7, are required to be zero at any call/return point.
- When programming in assembly, any modified L registers must be reset to zero before calling another function or returning to a calling function. (The compiler will handle this automatically for C/C++ code).
- Interrupt routines must save and set to zero the L register before using its corresponding I register for any post-modify instruction.
- The MMASK register ensures that MODE1 is set to the correct value before the interrupt dispatcher code is executed. It ensures that the following bits are cleared: BRO, BR8, IRPTEN, ALUSAT, PEYEN, BDCST1, BDCST9.

User Registers

The `-reserve` command-line switch lets you reserve registers for your inline assembly code or assembly language routines. If reserving an L register, you must reserve the corresponding I register; reserving an L register without reserving the corresponding I register can result in execution problems.

You must reserve the same list of registers in all linked files; the whole project must use the same `-reserve` option. [Table 1-30](#) lists these registers. Note that the C run-time library does not use these registers.



Reserving registers can negatively influence the efficiency of compiled C/C++ code; use this option sparingly.

Table 1-30. User Registers

Register	Value	Modification Rule
i0, b0, l0, m0, i1, b1, l1, m1, i8, b8, l8, m8, i9, b9, l9, m9, mrb, ustat1, ustat2, ustat3, ustat4	user defined	If not reserved, modify for temporary use, restore when done If reserved, usage is not limited

Call Preserved Registers

The cc21k C/C++ run-time environment specifies a set of registers whose contents must be saved and restored. Your assembly function must save these registers during the function's prologue and restore the registers as part of the function's epilogue. These registers must be saved and restored if they are modified within the assembly function; if a function does not change a particular register, it does not need to save and restore the register.

Table 1-31 lists these registers.

Table 1-31. Call Preserved Registers¹

b0	b1	b2	b3	b5	b8
b9	b10	b11	b14	b15	
i0	i1	i2	i3	i5	i8
i9	i10	i11	i14	i15	model
mode2	mrb	mrf	m0	m1	m2
m3	m8	m9	m10	m11	r3
r5	r6	r7	r9	r10	r11
r13	r14	r15			

- ¹ If you use a call preserved I register in an assembler routine called from an assembler routine, you must save and zero (clear) the corresponding L register as part of the function prologue. Then, restore the L register as part of the function epilogue.

C/C++ Run-Time Model and Environment

Many functions in the C/C++ run-time library expect the processor to be in a specific mode and may not operate correctly if the processor is in a different mode. If you need to change processor modes, save the old values in the `mode1` and `mode2` registers and restore these registers before calling or returning to calling functions.

The C/C++ run-time environment:

- Uses default bit order for `DAG` operations (no bit reversal)
- Uses the primary register set (not background set)
- Uses `.PRECISION=32` (32-bit floating-point) and `.ROUND_NEAREST` (round-to-nearest value) assembly directives
- Disables ALU saturation (`MODE1` register, `ALUSAT` bit = 0)
- Uses default `FIX` instruction rounding to nearest (`MODE1` register, `TRUNCATE=0`)
- Enables circular buffering on ADSP-2116x, ADSP-2126x, ADSP-2136x and ADSP-2137x processors by setting `CBUFEN` on `MODE1`

Scratch Registers

The `cc21k` C/C++ run-time environment specifies a set of registers whose contents do not need to be saved and restored. Note that the contents of these registers are not preserved across function calls. [Table 1-32](#) lists these registers.

Table 1-32. Scratch Registers

b4	b12	b13	r0	r1	r2	r4	r8	r12
i4	i12	m4	m12	i13	PX	USTAT1	USTAT2	

In addition, for ADSP-2116x processors, the PE_y data registers are all scratch registers. [Table 1-33](#) lists these registers.

Table 1-33. Additional ADSP-2116x Scratch Registers

s0	s1	s2	s3	s4	s5	s6	s7	s8
s9	s10	s11	s12	s13	s14	s15	USTAT3	USTAT4
ASTAT _y	STK _y							



The USTAT registers are now treated as scratch registers.

Stack Registers

The cc21k C/C++ run-time environment reserves a set of registers for controlling the run-time stack. These registers may be modified for stack management, but they must be saved and restored. [Table 1-34](#) lists these registers.

Table 1-34. Pointer Registers

Register	Value	Modification Rules
i7	Stack pointer	Modify for stack management, restore when done
i6	Frame pointer	Modify for stack management, restore when done
i12	Return address	Load with function call return address on function exit

Alternate Registers

The C/C++ run-time environment model does not use any of the alternate registers because these registers are available for use in assembly language only. To use these registers, several aspects of the C/C++ run-time model must be understood.

The C/C++ run-time model uses register I6 as the frame pointer and register I7 as the stack pointer. Setting the DAG register that contains I6 and I7 from a background register to an active register directly affects the stack operation. The C/C++ run-time model does not have an understanding of background registers.

If the background I6 and I7 registers are active and an interrupt occurs, the C/C++ run-time model still uses I6 and I7 to update the stack. This results in faulty stack handling.



The background register set containing DAG registers I6 and I7 should only be used in assembly routines if interrupts are not enabled.

The super-fast interrupt dispatcher uses context switching rather than saving registers on the run-time stack. To ensure no register conflicts, do not use the super fast interrupt dispatcher or disable interrupts when using secondary registers in an assembly routine.

Managing the Stack

The cc21k C/C++ run-time environment uses the run-time stack for storage of automatic variables and return addresses. The stack is managed by a frame pointer (FP) and a stack pointer (SP) and grows downward in memory, moving from higher to lower addresses.

A stack frame is a section of the stack used to hold information about the current context of the C/C++ program. Information in the frame includes local variables, compiler temporaries, and parameters for the next function.

The frame pointer serves as a base for accessing memory in the stack frame. Routines refer to locals, temporaries, and parameters by their offset from the frame pointer.

Figure 1-3 shows an example section of a run-time stack. In the figure, the currently executing routine, `Current()`, was called by `Previous()`, and `Current()` in turn calls `Next()`. The state of the stack is as if `Current()` has pushed all the arguments for `Next()` onto the stack and is just about to call `Next()`.



Stack usage for passing any or all of a function's arguments depends on the number and types of parameters to the function.

The prototypes for the functions in Figure 1-3 are as follows:

```
void Current(int a, int b, int c, int d, int e);
void Next(int v, int w, int x, int y, int z);
```

In generating code for a function call, the compiler produces the following operations to create the called function's new stack frame:

- Loads the `r2` register with the frame pointer (in the `i6` register)
- Sets the FP, `i6` register, equal to the SP (in the `i7` register)
- Uses the delayed-branch instruction to pass control to the called function
- Pushes the FP, `r2`, onto the run-time stack during the first branch delay slot
- Pushes the return address, `pc`, onto the run-time stack during the second delay-branch slot

For the ADSP-21020 processor, the following instructions create a new stack frame.

```
r2=i6;
i6=i7;
jump my_function (DB);
    /* where my_function is the called function */
dm(i7, m7) = r2;
dm(i7, m7) = pc;
```

C/C++ Run-Time Model and Environment

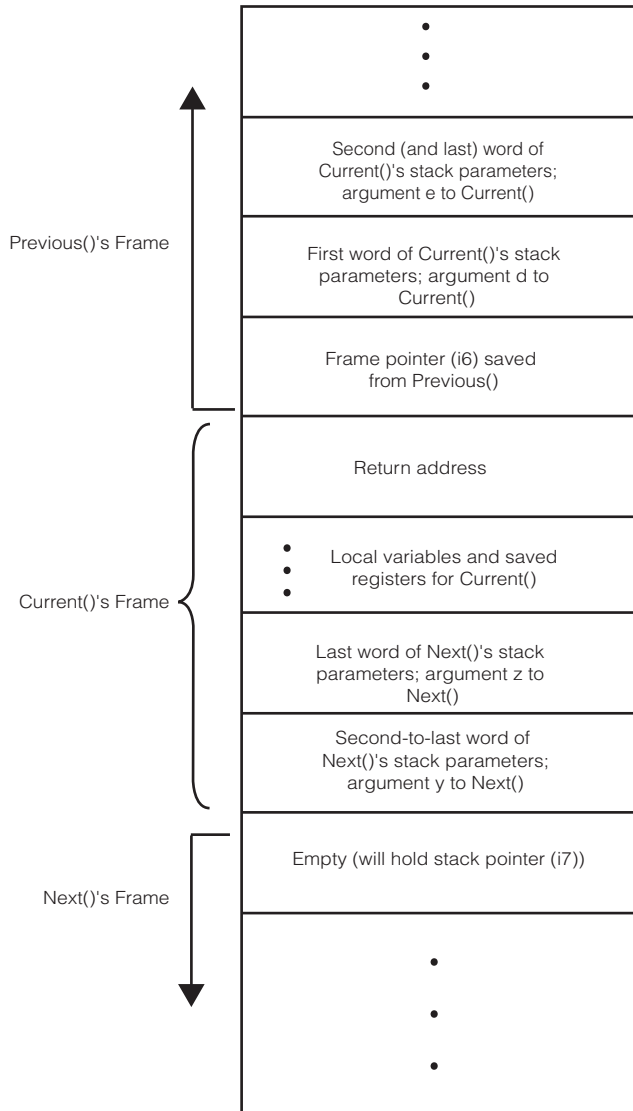


Figure 1-3. Example Run-Time Stack

For ADSP-2106x/2116x/2126x/2136x processors, the following instructions create a new stack frame. Note how the two initial register moves are incorporated into the `cjump` instruction.

```
cjump my_function (DB);
    /* where my_function is the called function */
dm(i7, m7) = r2;
dm(i7, m7) = pc;
```

As you write assembly routines, note that the operations to create a stack frame are the responsibility of the called function, and you can use the `entry` or `leaf_entry` macros to perform these operations. For more information on these macros, see [“Using Mixed C/C++ and Assembly Support Macros” on page 1-316](#).

In generating code for a function return, the compiler uses the following operations to restore the calling function’s stack frame.

- Pops the return address off the run-time stack and loads it into the `i12` register
- Uses the delayed-branch instruction to pass control to the calling function and jumps to the return address (`i12 + 1`)
- Restores the caller’s stack pointer, `i7` register, by setting it equal to `FP`, `i6` register, during the first branch delay slot
- Restores the caller’s frame pointer, `i6` register, by popping the previously saved `FP` off the run-time stack and loading the value into `i6` during the second delay-branch slot

For ADSP-2106x/2116x/2126x/2136x processors, the following instructions return from the function and restore the stack and frame pointers. Note that the restoring of `SP` and `FP` are incorporated into the `rframe` instruction.

```
i12 = dm(-1, i6);
jump (m14, i12) (DB);
nop;
rframe;
```

As you write assembly routines, note that the operations to restore stack and frame pointers are the responsibility of the called function, and you can use the `exit` or `leaf_exit` macros to perform these operations. For more information on these macros, see [“Using Mixed C/C++ and Assembly Support Macros” on page 1-316](#).

In the following code examples ([Listing 1-2](#) and [Listing 1-3](#)), observe how the function calls in the C code translate to stack management tasks in the compiled (assembly) version of the code. The comments have been added to the compiled code to indicate the function prologue and function epilogue.

Listing 1-2. Stack Management, Example C Code

```
/* Stack management - C code */

int my_func(int, int);
int arg_a, return_c;

main()
{
    static int arg_b;
    arg_b = 0;
    return_c = my_func(arg_a, arg_b);
}

int my_func(int arg_1, int arg_2);
{
    return (arg_1 + arg_2)/2;
}
```

Listing 1-3. Stack Management, Example ADSP-2106x Assembly Code

```
/* Stack management - C compiled (2106x assembly) code */
.section /pm seg_pmco;
.global _main;
_main:
```

```

        .def end_prologue; .val .; .scl 109; .endef;
r4=dm(_arg_a);
/* r4, the first argument register, which is arg_a */
r8=0;
/* r8, the second argument register, which is arg_b */
dm(arg_b)=r8;

/* The next three lines are the function call sequence */
cjump (pc,_my_func) (DB);
dm(i7,m7)=r2;
dm(i7,m7)=pc;

dm(_return_c)=r0;

/* The next four lines are main's function epilogue */
i12=dm(-1,i6);
jump (m14, i12) (DB);
nop;
rframe;

.global _my_func;
_my_func:
    .def end_prologue; .val .; .scl 109; .endef;
    r0=(r4+r8)/2;

/* The next four lines are my_func's function epilogue */
i12=dm(-1,i6);
jump (m14, i12) (DB);
nop;
rframe;

```

The next two sections, [“Transferring Function Arguments and Return Value” on page 1-304](#) and [“Using Macros to Manage the Stack” on page 1-330](#), provide additional detail on function call requirements.

Transferring Function Arguments and Return Value

The C/C++ run-time environment uses a set of registers and the run-time stack to transfer function parameters to assembly routines. Your assembly language functions must follow these conventions when they call or when they are called by C/C++ functions.

Because it is most efficient to use registers for passing parameters, the run-time environment attempts to pass the first three parameters in a function call using registers; it then passes any remaining parameters on the run-time stack.

The convention is to pass the function's first parameter in `r4`, the second parameter in `r8`, and the third parameter in `r12`. The following exceptions apply to this convention:

- If any parameter is larger than a single 32-bit word, then that parameter and all subsequent parameters are passed on the stack.
- If the function is declared to take a variable number of arguments (has `...` in its prototype), then the last named parameter and any subsequent parameters are passed on the stack.

Table 1-35 lists the rules that `cc21k` uses for passing parameters in registers to functions and the rules that your assembly code must use for returns.

Table 1-35. Parameter and Return Value Transfer Registers

Register	Parameter Type Passed Or Returned
<code>r4</code>	Pass first 32-bit data type parameter
<code>r8</code>	Pass second 32-bit data type parameter
<code>r12</code>	Pass third 32-bit data type parameter
<code>stack</code>	Pass fourth and remaining parameters; see exceptions to this rule on this page.
<code>r0</code>	Return <code>int</code> , <code>long</code> , <code>char</code> , <code>float</code> , <code>short</code> , <code>pointer</code> , and one-word structure parameters

Table 1-35. Parameter and Return Value Transfer Registers (Cont'd)

Register	Parameter Type Passed Or Returned
r0, r1	Return long double and two-word structure parameters. Place MSW in r0 and LSW in r1
r1	Return the address of results that are longer than two words; r1 contains the first location in the block of memory containing the results

Consider the following function prototype example.

```
pass(int a, float b, char c, float d);
```

The first three arguments, *a*, *b*, and *c* are passed in registers *r4*, *r8*, and *r12*, respectively. The fourth argument, *d*, is passed on the stack.

This next example illustrates the effects of passing doubles.

```
count(int w, long double x, char y, float z);
```

The first argument, *w*, is passed in *r4*. Because the second argument, *x*, is a multi-word argument, *x* is passed on the stack. As a result, the remaining arguments, *y* and *z*, are also passed on the stack.

The following example illustrates the effects of variable arguments on parameter passing.

```
compute(float k, int l, char m,...);
```

Here, the first two arguments, *k* and *l*, are passed in registers *r4* and *r8*. Because *m* is the last named argument, *m* is passed on the stack, as are all remaining variable arguments.

When arguments are placed on the stack, they are pushed on from right to left. The right-most argument is at a higher address than the left-most argument passed on the stack.

C/C++ Run-Time Model and Environment

The following example shows how to access parameters passed on the stack.

```
tab(int a, char b, float c, int d, int e, long double f);
```

Parameters `a`, `b`, and `c` are passed in registers because they are single-word parameters. The remaining parameters, `d`, `e`, and `f`, are passed on the stack.

All parameters passed on the stack are accessed relative to the frame pointer, register `i6`. The first parameter passed on the stack, `d`, is at address `i6 + 1`. To access it, you could use this assembly language statement.

```
r3=dm(1,i6);
```

The second parameter passed on the stack, `e`, is at `i6 + 2` and can be accessed by the statement

```
r3=dm(2,i6);
```

The third parameter passed on the stack, `f`, is a `long double` that has its most significant word at `i6 + 3` and its least significant word at `i6 + 4`. The most significant word of `f` can be accessed by the statement

```
r3=dm(3,i6);
```

Passing a C++ Class Instance

A C++ class instance function parameter is always passed by reference when a copy constructor has been defined for the C++ class. If a copy constructor has not been defined for the C++ class then the C++ class instance function parameter is passed by value.

Consider the following example.

```
class fr
{
    public:
```

```

int v;
public:
    fr () {}
    fr (const fr& rc1) : v(rc1.v) {}
};

extern int fn(fr x);

fr Y;

int main()
{
    return fn (Y);
}

```

The function call `fn (Y)` in `main` will pass the C++ class instance `Y` by reference because a copy constructor for that C++ class has been defined by `fr (const fr& rc1) : v(rc1.v) {}`. If this copy constructor were removed, then `Y` would be passed by value.

Using Data Storage Formats

The C/C++ run-time environment uses the data formats that appear in the [Table 1-36](#), [Table 1-37](#), [Figure 1-4 on page 1-308](#), and [Figure 1-5 on page 1-309](#).

Table 1-36. Data Storage Formats and Data Type Sizes

Applied Type	Number Representation
int	32-bit two's complement
long int	32-bit two's complement
short int	32-bit two's complement
unsigned int	32-bit unsigned magnitude
unsigned long int	32-bit unsigned magnitude
char	32-bit two's complement

Table 1-36. Data Storage Formats and Data Type Sizes (Cont'd)

Applied Type	Number Representation
unsigned char	32-bit unsigned magnitude
float	32-bit IEEE single-precision
double	32-bit IEEE single-precision or 64-bit IEEE double-precision if you compile with the -double-size-64 switch
long double	64-bit IEEE double-precision

Table 1-37. Data Storage Formats and Data Storage

Data	Big Endian Storage Format
long double	Writes 64-bit IEEE double-precision data with the most significant word closer to address 0x0000, proceeds toward the top of memory with the rest. (See Figure 1-5 for details.)

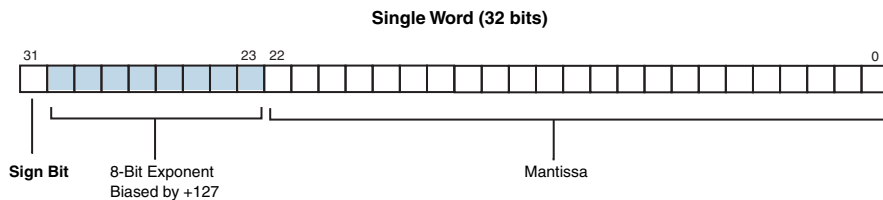


Figure 1-4. Data Storage Format for Float and Double Types

In [Figure 1-4](#) above the single word (32-bit) data storage format equates to:

$$-1^{Sign} \times 1.Mantissa \times 2^{(Exponent - 127)}$$

where

- Sign – Comes from the sign bit.
- Mantissa – Represents the fractional part of the mantissa 23 bits. (The “1.” is assumed in this format.)
- Exponent – Represents the 8-bit exponent.

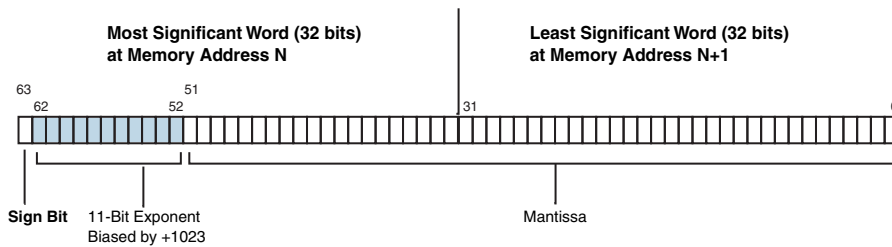


Figure 1-5. Double-Precision IEEE Format

In [Figure 1-5](#) above the two-word (64-bit) data storage format equates to:

$$-1^{Sign} \times 1.Mantissa \times 2^{(Exponent - 1023)}$$

where

- Sign – Comes from the sign bit.
- Mantissa – Represents the fractional part of the mantissa 52 bits. (The “1.” is assumed in this format.)
- Exponent – Represents the 11-bit exponent.

Using the Run-Time Header

The run-time header is an assembly language procedure that initializes the processor and sets up processor features to support the C/C++ run-time environment. The source code for the default run-time headers is in:

- `020_hdr.asm` for ADSP-21020 processors
- `06x_hdr.asm` for ADSP-2106x processors
- `16x_hdr.asm` for ADSP-2116x processors
- `26x_hdr.asm` for ADSP-2126x processors
- `36x_hdr.asm` for ADSP-2136x processors
- `37x_hdr.asm` for ADSP-2137x processors
- `214xx_hdr.asm` for ADSP-2146x processors

This run-time header performs the following operations:

- Initializes the C/C++ run-time environment
- Sets up the interrupt table
- Calls your `main()` routine

C/C++ and Assembly Interface

This section describes how to call assembly language subroutines from within C/C++ programs and how to call C/C++ functions from within assembly language programs.



Before attempting to do either of these calls, be sure to familiarize yourself with the information about the C/C++ run-time model (including details about the stack, data types, and how arguments are handled) in [“C/C++ Run-Time Environment” on page 1-267](#). At the end of this reference, a series of examples demonstrate how to mix C/C++ and assembly code.

This section includes:

[“Calling Assembly Subroutines from C/C++ Programs” on page 1-312](#)

[“Calling C/C++ Functions from Assembly Programs” on page 1-314](#)

[“C++ Programming Examples” on page 1-325](#)

[“Mixed C/C++/Assembly Programming Examples” on page 1-328](#)

[“Exceptions Tables in Assembly Routines” on page 1-340](#)

Calling Assembly Subroutines from C/C++ Programs

Before calling an assembly language subroutine from a C/C++ program, create a prototype to define the arguments for the assembly language subroutine and the interface from the C/C++ program to the assembly language subroutine. Even though it is legal to use a function without a prototype in C/C++, prototypes are a strongly-recommended practice for good software engineering. When the prototype is omitted, the compiler cannot perform argument-type checking and assumes that the return value is of type integer and uses K&R promotion rules instead of ANSI promotion rules.

The compiler prefaces the name of any external entry point with an underscore. You should either declare your assembly language subroutine's name with a leading underscore or define it within an `'extern "asm" {}'` format to tell the compiler that it is an assembly language subroutine.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated* registers. The scratch registers can be used within the assembly language program without worrying about their previous contents. If more registers are needed (or you work with existing code and wish to use the preserved registers), you must first save their contents and then restore those contents before returning. Do not use the dedicated registers for other than their intended purpose; the compiler, libraries, debugger, and interrupt routines all depend on having a stack available as defined by those registers.

The compiler also assumes that the machine state does not change during execution of the assembly language subroutine.



Do not change any machine modes; for example, the machine may have an integer/fractional mode, or it may use certain registers to indicate circular buffering when those register values are non-zero.

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer.

A good way to explore how arguments are passed between a C/C++ program and an assembly language subroutine is to write a dummy function in C/C++ and compile it with the `-save-temps` command-line switch. The following example includes the global volatile variable assignments to indicate where the arguments can be found upon entry to `asmfunc`.

```
// Sample file for exploring compiler interface ...
// global variables ... assign arguments there just so
// we can track which registers were used
// (type of each variable corresponds to one of arguments):

int global_a;
float global_b;
int * global_p;

// the function itself:

int asmfunc(int a, float b, int * p)
{
    // do some assignments so .s file will show where args are:
    global_a = a;
    global_b = b;
    global_p = p;

    // value gets loaded into the return register:
    return 12345;
}
```

When compiled with the switches `-save-temps -O0 -no-annotate`, this produces the following code.

```
_asmfunc;
.LN_asmfunc:
    modify(i7,-3);
    dm(-8,i6)=r12;
    dm(-3,i6)=r8;
```

```
dm(-4,i6)=r4;  
dm(_global_a)=r4;  
dm(_global_b)=r8;  
dm(_global_p)=r12;  
r0=12345;  
i12=dm(m7,i6);  
jump (m14,i12) (db); rframe; nop;  
.LN._asmfunc.end;  
._asmfunc.end;
```

Calling C/C++ Functions from Assembly Programs

You may want to call C/C++-callable library and other functions from within an assembly language program. As discussed in [“Calling Assembly Subroutines from C/C++ Programs” on page 1-312](#), you may wish to create a test function to do this in C/C++, and then use the code generated by the compiler as a reference when creating your assembly language program and the argument setup. Using volatile global variables may help clarify the essential code in your test function.

The run-time model defines some registers as *scratch* registers and others as *preserved* or *dedicated*. The contents of the scratch registers may be changed without warning by the called C/C++ function; if the assembly language program needs the contents of any of those registers, you *must* first *save* their contents before the call to the C/C++ function and then *restore* those contents after returning from the call.



Do *not* use the dedicated registers for other than their intended purpose; the compiler, libraries, debugger and interrupt routines all depend on having a stack available as defined by those registers.

Preserved registers can be used; their contents are not changed by calling a C/C++ function. The function always saves and restores the contents of preserved registers if they are going to change.

If arguments are on the stack, they are addressed via an offset from the stack pointer or frame pointer. You can explore how arguments are passed between an assembly language program and a function by writing a dummy function in C/C++ and compiling it with the `save temporary files` option (the `-save-temps` command-line switch). By examining the contents of volatile global variables in `*.s` file, you can determine how the C/C++ function passes arguments and then duplicate that argument setup process in the assembly language program.

The stack must be set up correctly before calling a C/C++-callable function. If you call other functions, maintaining the basic stack model also facilitates the use of the debugger. The easiest way to do this is to define a C/C++ main program to initialize the run-time system; hold the stack until it is needed by the C/C++ function being called from the assembly language program; and then hold that stack until it is needed to call back into C/C++, making sure the dedicated registers are correct. You do not need to set the `FP` prior to the call; the caller's `FP` is never used by the callee.

The following example demonstrates the features described in this section. Because so many different features are combined into a single example, this procedure as a whole should not be viewed as an example of good assembly programming.

```
// PROCEDURE: memalloc
.global _memalloc;
_memalloc:
    r5=0xffff;           // Assign a value to preserved reg r5
    r8=0xffff;           // Assign a value to scratch reg r8

    r0=dm(-3,i6);        // Read a value from the stack
    r4=r0;               // Put this value in a parameter register

    // Save value of scratch register prior to function call
    r7=r8;

    // Call the C function malloc()
```

C/C++ and Assembly Interface

```
r2=i6;
i6=i7;
jump _malloc (DB);
dm(i7,m7)=r2;
dm(i7,m7)=pc;

// Check the result of the function call
r0=pass r0;
if eq jump(pc,_error);

// Check that the preserved register did not change over
// the function call
r4=0xffff;
comp(r4,r5);
if ne jump(pc, _error);

// Restore value of scratch register after function call
r8=r7;

i6 = 0x123;          // PROGRAMMING ERROR! Do not change
                     // dedicated registers

rts;
```

Using Mixed C/C++ and Assembly Support Macros

This section lists C/C++ and assembly interface support macros in the `asm_sprt.h` system header file. Use these macros for interfacing assembly language modules with C/C++ functions. [Table 1-38](#) lists the macros.



Although the syntax for each macro does not change, the listing of `asm_sprt.h` in this section may not be the most recent version. To see the current version, check the `asm_sprt.h` file that came with your software package.

The following are the descriptions and the syntax for the C/C++ and assembly interface support macros.

Table 1-38. Interface Support Macro Summary

<code>entry</code>	<code>exit</code>	<code>leaf_entry</code>	<code>leaf_exit</code>
<code>ccall(x)</code>	<code>reads(x)</code>	<code>puts</code>	<code>gets(x)</code>
<code>alter(x)</code>	<code>save_reg</code>	<code>restore_reg</code>	

entry

The `entry` macro expands into the function prologue for non-leaf functions. This macro should be the first line of any non-leaf assembly routine. Note that this macro is currently null, but it should be used for future compatibility.

exit

The `exit` macro expands into the function epilogue for non-leaf functions. This macro should be the last line of any non-leaf assembly routine. Exit is responsible for restoring the caller's stack and frame pointers and jumping to the return address. Note that this macro is currently null, but it should be used for future compatibility.

leaf_entry

The `leaf_entry` macro expands into the function prologue for leaf functions. This macro should be the first line of any non-leaf assembly routine. Note that this macro is currently null, but it should be used for future compatibility.

leaf_exit

The `leaf_exit` macro expands into the function epilogue for leaf functions. This macro should be the last line of any leaf assembly routine. `leaf_exit` is responsible for restoring the caller's stack and frame pointers and jumping to the return address.

ccall(x)

The `ccall` macro expands into a series of instructions that save the caller's stack and frame pointers and then jump to function `x()`.

reads(x)

The `reads` macro expands into an instruction that reads a value off the stack and puts the value in the indicated register.

puts=x

The `puts` macro expands into an instruction that pushes the value in register `x` onto the stack.

gets(x)

The `gets` macro expands into an instruction that reads a value off the stack and puts the value in the indicated register.

```
register = gets(x);
```

The value is located at an offset `x` from the stack pointer.

alter(x)

The `alter` macro expands into an instruction that adjusts the stack pointer by adding the immediate value `x`. With a positive value for `x`, `alter` pops `x` words from the top of the stack. You could use `alter` to clear `x` number of parameters off the stack after a call.

save_reg

The `save_reg` macro expands into a series of instructions that push the register file registers (`r0-r15`) onto the run-time stack.

restore_reg

The `restore_reg` macro expands into a series of instructions that pop the register file registers (r0-r15) off the run-time stack.

The following code example shows the `asm_sprt.h` used for defining C/C++/assembly interface support macros.

```
/* asm_sprt.h - C/C++/Assembly Interface Support Macros */
/* asm_sprt.h - $Date: 10/09/97 6:28p $ */

#ifndef __ASM_SPRT_DEFINED
#define __ASM_SPRT_DEFINED

#define entry /* nothing */
#define leaf_entry /* nothing */

#ifdef __ADSP21020__
#define ccall(x) \
    r2=i6; i6=i7; \
    jump (pc, x) (db); \
    dm(i7,m7)=r2; \
    dm(i7,m7)=PC;
#define leaf_exit \
    i12=dm(m7,i6); \
    jump (m14,i12) (db); \
    i7=i6; i6=dm(0,I6);
#define exit leaf_exit
#else
#define ccall(x) \
    cjump (x) (DB); \
    dm(i7,m7)=r2; \
    dm(i7,m7)=PC;

#define leaf_exit \
    i12=dm(m7,i6); \
    jump (m14,i12) (db); \
    nop; \
    RFRAME
#endif
```

C/C++ and Assembly Interface

```
#define exit leaf_exit
#endif

#define reads(x)dm(x, i6)
#define putsdm(i7, m7)
#define gets(x)dm(x, i7)
#define alter(x)modify(i7, x)

#define save_reg \
    puts=r0;\
    puts=r1;\
    puts=r2;\
    puts=r3;\
    puts=r4;\
    puts=r5;\
    puts=r6;\
    puts=r7;\
    puts=r8;\
    puts=r9;\
    puts=r10;\
    puts=r11;\
    puts=r12;\
    puts=r13;\
    puts=r14;\
    puts=r15;

#define restore_reg \
    r15=gets(1);\
    r14=gets(2);\
    r13=gets(3);\
    r12=gets(4);\
    r11=gets(5);\
    r10=gets(6);\
    r9 =gets(7);\
    r8 =gets(8);\
    r7 =gets(9);\
    r6 =gets(10);\
    r5 =gets(11);\
    r4 =gets(12);\
    r3 =gets(13);\
```



```

    r2 =gets(14);\
    r1 =gets(15);\
    r0 =gets(16);\
    alter(16);

#endif

```

Using Mixed C/C++ and Assembly Naming Conventions

It is necessary to be able to use C/C++ symbols (function or variable names) in assembly routines and use assembly symbols in C/C++ code. This section describes how to name and use C/C++ and assembly symbols.

To name an assembly symbol that corresponds to a C/C++ symbol, add an underscore prefix to the C/C++ symbol name when declaring the symbol in assembly. For example, the C/C++ symbol `main` becomes the assembly symbol `_main`.

To use a C function or variable in an assembly routine, declare it as global in the C program. Import the symbol into the assembly routine by declaring the symbol with the `.EXTERN` assembler directive.

The external name of a C++ function encodes information about its type and parameters. Function “signature” enables the overloading of functions and operators that the C++ language supports. To reference a function in a C++ module, declare it with the `extern “C”` specifier in order to use the naming convention of C. Note that C++ data symbols use the same convention as C.

To use an assembly function or variable in your C program, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern` in the C program.

To use an assembly function in your C++ module, declare the symbol with the `.GLOBAL` assembler directive in the assembly routine and import the symbol by declaring the symbol as `extern “C”` in the C++ program. For

C/C++ and Assembly Interface

example, to reference the `_funcmult` assembly routine from a C++ program, you declare it as `extern "C" int funcmult(int a, int b)` in the C++ program.

Table 1-39 shows several examples of the C/Assembly interface naming conventions. Each row shows how assembler code can reference the corresponding C item.

Table 1-39. C Naming Conventions For Symbols

In the C Program	In the Assembly Subroutine
<code>int c_var; /*declared global*/</code>	<code>.extern _c_var;</code>
<code>void c_func(){...}</code>	<code>.extern _c_func;</code>
<code>extern int asm_var;</code>	<code>.global _asm_var;</code>
<code>extern void asm_func();</code>	<code>.global _asm_func;</code> <code>_asm_func:</code>

Table 1-40 shows several examples of the C++/Assembly interface naming conventions. Each row shows how assembler code can reference the corresponding C++ item.

Table 1-40. C++ Naming Conventions for Symbols

In the C++ Program	In the Assembly Subroutine
<code>extern "C" { int c_var; } /*declared global*/</code>	<code>.extern _c_var;</code>
<code>extern "C" void c_func(void){...}</code>	<code>.extern _c_func;</code>
<code>extern "C" int asm_var;</code>	<code>.global _asm_var;</code>
<code>extern "C" void asm_func(void);</code>	<code>.global _asm_func;</code> <code>_asm_func:</code>

Implementing C++ Member Functions in Assembly Language

If an assembly language implementation is desired for a C++ member function, the simplest way is to use C++ to provide the proper interface between C++ and assembly.

In the class definition, write a simple member function to call the assembly-implemented function (subroutine). This call can establish any interface between C++ and assembly, including passing a pointer to the class instance. Since the call to the assembly subroutine resides in the class definition, the compiler inlines the call (inlining adds no overhead to compiler performance). From an efficiency point of view, the assembly language function is called directly from the user code.

As for any C++ function, ensure that a prototype for the assembly-implemented function is included in your program. As discussed in [“Using Mixed C/C++ and Assembly Naming Conventions” on page 1-321](#), you declare your assembly language subroutine’s name with the `.GLOBAL` directive in the assembly portion and import the symbol by declaring it as `extern “C”` in the C++ portion of the code.

Note that using this method you avoid name mangling—you choose your own identifier for the external function. Access to internal class information can be done either in the C++ portion or in the assembly portion. If the assembly subroutine needs only limited access to the class members, it is easier to select those in the C++ code and pass them as explicit arguments. This way the assembly code does not need to know how data is allocated within a class instance.

```
#include <stdio.h>
/* Prototype for external assembly routine: */
/* C linkage does not have name mangling */
extern "C" int cc_array(int);

class CC {
private:
    int av;
public:
    CC(){};
    CC(int v) : av(v){};
    int a() {return av;};
                                /* Assembly routine call: */
    int array() {return cc_array(av);};
};
```

C/C++ and Assembly Interface

```
};

int main()
{
    CC samples(11);
    CC points;
    points = CC(22);
    int j, k;
    j = samples.a();
    k = points.array();    // Test asm call


    printf ( "Val is %d\n", j );
    printf ( "Array is %d\n", k );

    return 1;
}
/* In a separate assembly file: */
.section /pm seg_pmco;
.global _cc_array;
_cc_array:
modify(i7,-3);
dm(-4,i6)=r3;
dm(-2,i6)=r4;
r3=r4;
r0=r3+r3;
r3=dm(-4,i6);
i12=dm(m7,i6);
jump(m14,i12)(DB);
rframe;
nop;
```

Writing C/C++ Callable SIMD Subroutines

You can write assembly subroutines that use SIMD mode for the ADSP-2116x, ADSP-2126x and ADSP-213xx processors and call them from your C programs. The routine may use SIMD mode (PEYEN bit=1)

for all code between the function prologue and epilogue, placing the chip in SISD mode (`PEYEN` bit =0) before the function epilogue or returning from the function.

 While it is possible to write subroutines that can be called in SIMD mode (the chip is in SIMD mode before the call and after the return), the compiler does not support a SIMD call interface at this time. For example, trying to call a subroutine from a `#pragma SIMD_for` loop prevents the compiler from executing the loop in SIMD mode because the compiler does not support SIMD mode calls.

Because transfers between memory and data registers are doubled in SIMD mode (each explicit transfer has a matching implicit transfer), it is recommended that you access the stack in SISD mode to prevent corrupting the stack. For more information on SIMD mode memory accesses, see the *Memory* chapter in the hardware reference for the appropriate ADSP-211xx, ADSP-212xx or ADSP-213xx processor.

If you are using SIMD subroutines, your interrupt handler must provide additional support. This support in the interrupt service routine entails saving-restoring the `PEYEN` bit and placing the processor in the mode (SISD or SIMD) that the interrupt service routine needs. Interrupt handlers often use the `MMASK` register to expedite these mode changes.

C++ Programming Examples

This section provides the following examples for C++-specific features:

- [“Using Fract Support”](#)
- [“Using Complex Support”](#)

Note that the `cc21k` compiler runs in C mode by default. To run the compiler in C++ mode, select the corresponding option on the command line, or check it in the **Project Options** dialog box of the VisualDSP++ environment.

For example, the following command line

```
cc21k -c++ fdot.c -T060.ldf
```

runs cc21k with:

-c++	Specifies that the following source file is written in ANSI/ISO standard C++ extended with the Analog Devices keywords.
fdot.c	Specifies the source file for your program.
-T 060.ldf	Specifies the Linker Description File for the ADSP-21060 processors.

Using Fract Support

[Listing 1-4](#) demonstrates the compiler support for the `fract` type and associated arithmetic operators, such as `+` and `*`. The dot product algorithm is expressed using the standard arithmetic operators. The code demonstrates how two variable-length arrays are initialized with fractional literals.

For more information about the fractional data type and arithmetic, see [“C++ Fractional Type Support” on page 1-228](#).

Listing 1-4. Dot Product Using Fract Arithmetic Example — C++ Code

```
#include <saturate.h>
#include <fract>
#define N 50
fract fdot (int array_size, fract *x, fract *y)
{
    int j;
    fract s;
    s = 0;
    for (j=0; j < array_size; j++)
    {
        s += x[j] * y[j];
    }
    return s;
}
```

```
extern fract a[N],b[N];
int main(void)
{
    set_saturate_mode();
    fdot (N,a,b);
}
```

Using Complex Support

The Mandelbrot fractal set is defined by the following iteration on complex numbers:

$$z := z * z + c$$

The c values belong to the set for which the above iteration does not diverge to infinity. The canonical set is defined when z starts from zero.

[Listing 1-5](#) demonstrates the Mandelbrot generator expressed in a simple algorithm using the C++ library `complex` class.

Listing 1-5. Mandelbrot Generator Example — C++ code

```
#include <complex>
int iterate (complex<double> c, complex<double> z, int max)
{
    int n;
    for (n = 0; n<max && abs(z)<2.0; n++)
    {
        z = z * z + c;
    }
    return (n == max ? 0 : n);
}
```

[Listing 1-6](#) shows a C version of the inner computational function of the Mandelbrot generator, which extracts performance and programming penalties (compared with the C++ version).

Listing 1-6. Mandelbrot Generator Example — C code

```
int    iterate (double creal, double cimag,
double zreal, double zimag, int max)
{
    double real, imag;
    int n;
    real = zreal * zreal;
    imag = zimag * zimag;
    for (n = 0; n<max && (real+imag)<5.0; n++)
    {
        zimag = 2.0 * zreal * zimag + cimag;
        zreal = real - imag + creal;
        real = zreal * zreal;
        imag = zimag * zimag;
    }
    return (n == max ? 0 : n);
}
```

Mixed C/C++/Assembly Programming Examples

This section shows examples of types of mixed C/C++/assembly programming in order of increasing complexity. The examples in this section are as follows:

- [“Using Inline Assembly \(Add\)”](#)
- [“Using Macros to Manage the Stack” on page 1-330](#)
- [“Using Scratch Registers \(Dot Product\)” on page 1-332](#)
- [“Using Void Functions \(Delay\)” on page 1-333](#)
- [“Using the Stack for Arguments \(Add 5\)” on page 1-334](#)
- [“Using Registers for Arguments and Return \(Add 2\)” on page 1-335](#)

- “Using Non-Leaf Routines That Make Calls (RMS)” on page 1-336
- “Using Call Preserved Registers (Pass Array)” on page 1-338

Note that leaf assembly routines are routines that return without making any calls. Non-leaf assembly routines call other routines before returning to the caller.

Note that you can use `cc21k` to compile your C/C++ program and assemble your assembly language modules. This ensures that the assembly of your modules complies with the C/C++ run-time environment.

For example, the following `cc21k` command line

```
cc21k my_prog.c my_sub1.asm -T 062.ldf -Wremarks
```

runs `cc21k` with the following modules listed in [Table 1-41](#):

Table 1-41. Modules for Running `cc21k`

Module	Description
<code>my_prog.c</code>	Selects a C language source file for your program
<code>my_sub1.asm</code>	Selects an assembly language module to be assembled and linked with your program
<code>-T 062.ldf</code>	Selects a Linker Description File describing your system
<code>-Wremarks</code>	Selects diagnostic compiler warnings

Using Inline Assembly (Add)

The following example shows how to write a simple routine in ADSP-21xxx assembly code that properly interfaces to the C/C++ environment. It uses the `asm()` construct to pass inline assembly code to the compiler.

```
int add(int x, int y, int z)
{
    int res;
    asm( "%0=%2+%1; %0=%0+%3":
        "=d"(res): "d"(x), "d"(y), "d"(z));
    return res;
}
```

Using Macros to Manage the Stack

[Listing 1-7](#) and [Listing 1-8 on page 1-331](#) show how macros can simplify function calls between C, C++, and assembly functions. The assembly function uses the `entry`, `exit`, and `ccall` macros to keep track of return addresses and manage the run-time stack. [For more information, see “Managing the Stack” on page 1-298.](#)

Listing 1-7. Subroutine Return Address Example — C Code

```
/* Subroutine Return Address Example—C code: */
/* assembly and c functions prototyped here */
void asm_func(void);
void c_func(void);

/* c_var defined here as a global */
/* used in .asm file as _c_var */
int c_var=10;

/* asm_var defined in .asm file as _asm_var */
```

```

extern int asm_var;

main ()
{
    asm_func();          /* call to assembly function      */
}

/* this function gets called from asm file */
void c_func(void)
{
    if (c_var != asm_var)
        exit(1);
    else
        exit(0);
}

```

Listing 1-8. Subroutine Return Address Example—Assembly Code

```

/* Subroutine Return Address Example—Assembly code:  */
#include <asm_sprt.h>
.section/dm seg_dmda;
.var _asm_var=0;          /* asm_var is defined here  */
.global _asm_var;         /* global for the C function */

.section/pm seg_pmco;
.global _asm_func;        /* _asm_func is defined here */
.extern _c_func;          /* c_func from the C file   */
.extern _c_var;           /* c_var from the C file    */

_asm_func:
entry;                    /* entry macro from asm_sprt */

    r8=dm(_c_var);        /* access the global C var  */
    dm(_asm_var)=r8;      /* set _asm_var to _c_var)  */

    ccall(_c_func);       /* call the C function      */

    exit;                 /* exit macro from asm_sprt */

```

Using Scratch Registers (Dot Product)

To write assembly functions that can be called from a C/C++ program, your assembly code must follow the conventions of the C/C++ run-time environment and use the conventions for naming functions. The `dot()` assembly function described below demonstrates how to comply with these specifications.

This function computes the dot product of two vectors. The two vectors and their lengths are passed as arguments. Because the function uses only scratch registers (as defined by the run-time environment) for intermediate values and takes advantage of indirect addressing, the function does not need to save or restore any registers.

```
/* dot(int n, dm float *x, pm float *y);
Computes the dot product of two floating-point vectors of length
n. One is stored in dm and the other in pm. Length n must be
greater than 2.*/

#include <asm_sprt.h>

.section/pm seg_pmco;
/* By convention, the assembly function name is the C function
name with a leading underscore; "dot()" in C becomes "_dot" in
assembly */

.global _dot;
_dot:

leaf_entry;

r0=r4-1,i4=r8;
/* Load first vector address into I register, and load r0 with
length -1 */

r0=r0-1,i12=r12;
```

```

/* Load second vector address into I register and load r0 with
length-2 (because the 2 iterations outside feed and drain the
pipe */

f12=f12-f12,f2=dm(i4,m6),f4=pm(i12,m14);
/* Zero the register that will hold the result and start
feeding pipe */

f8=f2*f4, f2=dm(i4,m6),f4=pm(i12,m14);
/* Second data set into pipeline, also do first multiply */

lcntr=r0, do dot_loop until lce;
/* Loop length-2 times, three-stage pipeline: read, mult, add */

dot_loop:
    f8=f2*f4, f12=f8+f12,f2=dm(i4,m6),f4=pm(i12,m14);
    f8=f2*f4, f12=f8+f12;
    f0=f8+f12;
/* drain the pipe and end with the result in r0, where it'll be
returned */

leaf_exit;
/* restore the old frame pointer and return */

```

Using Void Functions (Delay)

The simplest kind of assembly routine is one with no arguments and no return value, which corresponds to C/C++ functions prototyped as `void my_function(void)`. Such routines could be used to monitor an external event or used to perform an operation on a global variable.

It is important when writing such assembly routines to pay close attention to register usage. If the routine uses any call-preserved or compiler reserved registers (as defined in the run-time environment), the routine must save the register and restore it before returning. Because the following example does not need many registers, this routine uses only scratch registers (also defined in the run-time environment) that do not need to be saved.

C/C++ and Assembly Interface

Note that in the example all symbols that need to be accessed from C/C++ contain a leading underscore. Because the assembly routine name `_delay` and the global variable `_del_cycle` must both be available to C and C++ programs, they contain a leading underscore in the assembly code.

```
/* Simple Assembly Routines Example – _delay */
/* void delay (void);
An assembly language subroutine to delay N cycles, where N is
the value of the global variable del_cycle */

#include <asm_sprt.h>;

.section/pm seg_pmco;
.extern _del_cycle;
.global _delay;
_delay:
    leaf_entry;                /* first line of any leaf func */
    R4 = DM (_del_cycle);
/* Here, register r4 is used because it is a scratch register
and does not need to be preserved */
    LCNTR = R4, D0 d_loop UNTIL LCE;
    d_loop:
    nop;
    leaf_exit;                /* last line of any leaf func */
```

Using the Stack for Arguments (Add 5)

A more complicated kind of routine is one that has parameters but no return values. The following example adds together the five integers passed as parameters to the function.

```
/* Assembly Routines With Parameters Example – _add5 */
/* void add5 (int a, int b, int c, int d, int e);
An assembly language subroutine that adds 5 numbers */

#include <asm_sprt.h>
.section/pm seg_pmco;
.extern _sum_of_5; /* variable where sum will be stored */
.global _add5;
```

```

_add5:
leaf_entry;
/* the calling routine passes the first three parameters in
registers r4, r8, r12 */

r4 = r4 + r8;      /* add the first and second parameter */
r4 = r4 + r12;     /* adds the third parameter          */

/* the calling routine places the remaining parameters
(fourth/fifth) on the run-time stack; these parameters can be
accessed using the reads() macro */

r8 = reads(1);     /* put the fourth parameter in r8      */
r4 = r4 + r8;      /* adds the fourth parameter          */
r8 = reads(2);     /* put the fifth parameter in r8      */
r4 = r4 + r8;      /* adds the fifth parameter          */

dm(_sum_of_5) = r4;
/* place the answer in the global variable */

leaf_exit;

```

Using Registers for Arguments and Return (Add 2)

There is another class of assembly routines in which the routines have both parameters and return values. The following example of such a routine adds two numbers and returns the sum. Note that this routine follows the run-time environment specification for passing function parameters (in registers r4 and r8) and passing the return value (in register r0).

```

/* Routine With Parameters & Return Value -add2_ */
/* int add2 (int a, int b);
An assembly language subroutine that adds two numbers and returns
the sum */

#include <asm_sprt.h>

```

```
.section/pm seg_pmco;
.global _add2;
_add2:
leaf_entry;

/* per the run-time environment, the calling routine passes the
first two parameters passed in registers r4 and r8; the return
value goes in register r0 */

r0 = r4 + r8;
/* add the first and second parameter, store in r0*/

leaf_exit;
```

Using Non-Leaf Routines That Make Calls (RMS)

A more complicated example, which calls another routine, computes the root mean square of two floating-point numbers, such as

$$z = \sqrt{x^2 + y^2}$$

Although it is straight forward to develop your own function that calculates a square-root in ADSP-21xxx assembly language, the following example demonstrates how to call the square root function from the C/C++ run-time library, `sqrtf`. In addition to demonstrating a C run-time library call, this example shows some useful calling macros.

```
/* Non-Leaf Assembly Routines Example - _rms */
/* float rms(float x, float y); An assembly language subroutine
to return the rms z = (x^2 + y^2)^(1/2) */

#include <asm_sprt.h>
```



```

.section/pm seg_pmco;
.extern _sqrtf;
.global _rms;
_rms:
    entry;                /* first line of non-leaf routine */

    f4 = f4 * f4;
    f8 = f8 * f8;
    f4 = f4 + f8;
/* f4 contains argument to be passed to sqrtf function */

    ccall (_sqrtf);
/* use the ccall() macro to make a function call in a C
environment; f0 contains the result returned by the _sqrtf
function. In turn, _rms returns the result to its caller in f0
(and it is already there) */
    exit;                /* last line of non-leaf routine */

```

If a called function takes more than three single word parameters, the remaining parameters must be pushed on to the stack and popped off the stack after the function call. The following function could call the `_add5` routine shown in [“Using the Stack for Arguments \(Add 5\)” on page 1-334](#). Note that the last parameter must be pushed on the stack first.

```

/* Non-Leaf Assembly Routines Example – _calladd5 */
/* int calladd5 (void); An assembly language subroutine that
calls another routine with more than 3 parameters.
This example adds the numbers 1, 2, 3, 4, and 5. */

#include <asm_sprt.h>
.section/pm seg_pmco;
.extern _add5;
.extern _sum_of_5;
.global _calladd5;
_calladd5:

entry;

```

C/C++ and Assembly Interface

```
    r4 = 5;
/* the fifth parameter is stored in r4 for pushing onto stack */
    puts=r4;          /* put fifth parameter in stack */
    r4 = 4;
/* the fourth parameter is stored in r4 for pushing onto stack */
    puts=r4;          /* put fourth parameter in stack */
    r4 = 1;           /* the first parameter is sent in r4 */
    r8 = 2;           /* the second parameter is sent in r8 */
    r12 = 3;          /* the third parameter is sent in r12 */

    ccall (_add5);
/* use the ccall macro to make a function call in a C environment */
    alter(2);
/* call the alter() macro to remove the two arguments from
the stack */
    r0 = dm(_sum_of_5);
/* _sum_of_5 is where add5 stored its result */
    exit;
```

Using Call Preserved Registers (Pass Array)

Some functions need to make use of registers that the run-time environment defines as *call preserved registers*. These registers, whose contents are preserved across function calls, are useful for variables whose lifetime spans a function call. The following example performs an operation on the elements of a C array using call preserved registers.

```
/* Non-Leaf Assembly Routines Example - _pass_array */
/* void pass_array(
float function(float),
float *array,
int length);
An assembly language routine that operates on a C array */

#include <asm_sprt.h>
.section/pm seg_pmco;
.global _pass_array;
_pass_array:
    entry;
```

```

    puts = i8;
/* This function uses a call preserved register, i8, because
it could be used by multiple functions, and this way it does
not have to be stored for every function call */

    r0 = i1;
    puts = r0;      /* i1 is also call preserved */

i8 = r4;
/* read the first argument, the address of the function to call */

i1 = r8;
/* read the second argument, the C array containing the data
to be processed */

r0 = r12;
/* read third argument, the number of data points in the array */

lcntr=r0, do pass_array_loop until lce;
/* loop through data points */

f4=dm(i1,m5);
/* get data point from array, store it in f4 as a parameter for
the function call */

    r2=i6;
    i6=i7;
    jump (m13,i8) (DB);
    dm(i7,m7)=r2;
    dm(i7,m7)=PC;

pass_array_loop:
    dm(i1,m6)=f0;
/* store the return value back in the array */
    i1 = gets(1);      /* restore the value of i1 */
    i8 = gets(2);      /* restore the value of i8 */

```

```
exit;
```

Exceptions Tables in Assembly Routines

Assembly routines that both call C++ functions and are called by C++ functions, and require exceptions thrown by callees to be caught by callers need to be provided with a “function exceptions table” to enable the runtime library to restore registers to the values they held on entry to the routine.

The assembly routine must allocate a stack frame using `FP` and `SP` as described in [“Managing the Stack” on page 1-298](#). On entry to the assembly routine, call preserved registers ([on page 1-295](#)) that are modified in the routine should be saved into a contiguous region within the stack frame, called the save area. Registers are saved at ascending addresses in the save area in the order given in [Table 1-43 on page 1-341](#).

A word in the `.gdt` section must be initialised with the address of the function exceptions table, and the function exceptions table itself must be initialised as illustrated in [Table 1-42](#).

Table 1-42. Function Exceptions Table

Offset	Size	Meaning
0	1	Start address of the routine
1	1	First address after end of routine
2	1	Signed offset from FP of register save area
3	4	Bit set indicating which registers are saved
8	1	Always zero. Indicates this is not C++ code

The bit set field of the function exceptions table contains a bit for each register. The bits corresponding to registers saved in the save area must be set to one and the other bits set to zero. The bit numbers corresponding to each register are given in [Table 1-43](#), where bit 0 is the least significant bit of the lowest addressed word, bit 31 the most significant bit of that word, bit 32 the least significant bit of the second lowest addressed word and so on.

Bit numbering may best be explained by the C code to test bit number,

```
int wrd = r/32;
int bit = 1u << (r%32);
if (bitset[wrd] & bit)
    /* register r was saved */
```

Table 1-43. Function Exception Table Register Numbers

Register	Bit Number	Words taken in save area if saved
ASTAT	0	1
ASTATY	1	1
R0 - R15	2 - 17	1
S0 - S15	18 - 33	1
M0 - M15	34 - 49	1
B0 - B15	50 - 65	1

Table 1-43. Function Exception Table Register Numbers (Cont'd)

I0 - I15	66 - 81	1
L0 - L15	82 - 97	1
MRF	98	3
SMRF	99	3
MRB	100	3
SMRB	101	3
PX1, PX2	102 - 103	1
USTAT1 - USTAT4	104 - 107	1

This example shows an assembly routine with function exceptions table.

`.section/pm seg_pmco;`

```

_asmfunc:
.LN._asmfunc:
modify(i7,-6);           // allocate stack frame
                        // save area at I6-7
dm(-7,i6)=r5;           // save_area[0] = r5
dm(-6,i6)=r6;           // save_area[1] = r6
dm(-5,i6)=r7;           // save_area[2] = r7
r2=i0; dm(-4,i6)=r2;    // save_area[3] = i0
r2=i1; dm(-3,i6)=r2;    // save_area[4] = i1
r2=i2; dm(-2,i6)=r2;    // save_area[5] = i2
// use R5,R6,R7,I0,I1,I2, call a C++ function
i0=dm(-4,i6);
i1=dm(-3,i6);
i2=dm(-2,i6);
r5=dm(-7,i6);
r6=dm(-6,i6);
r7=dm(-5,i6);
i12=dm(m7,i6);
i12=dm(m7,i6);
jump (m14,i12) (db); rframe; nop;
.LN._asmfunc.end:
._asmfunc.end:
.global _asmfunc;

```

```

.type _asmfunc, STT_FUNC;
.section/dm .edt; // conventionally function exceptions
                  // tables go in .edt
.var .function_exceptions_table[8] =
.LN._asmfunc,      // first address of _asmfunc
.LN._asmfunc.end,  // first address after _asmfunc
-7,               // offset of save area from I6
0x000000380, 0, 0x0000001c, 0,
                // bit set, bits 7=R5,8=R6,9=R7,66=I0,67=I1,68=I2
0;               // always zero for non-c++
.section/dm .gdt;
.align 4;
.fet_index:
.var = .function_exceptions_table;
                // address of table in .gdt
.retain_name .fet_index;

```

Compiler C++ Template Support

The compiler provides template support C++ templates as defined in the ISO/IEC 14882:1998 C++ standard, with the exception that the `export` keyword is not supported.

Template Instantiation

Templates are instantiated automatically during compilation using a linker feedback mechanism. This involves compiling files, determining any required template instantiations, and then recompiling those files making the appropriate instantiations. The process repeats until all required instantiations have been made. Multiple recompilations may be required in the case when a template instantiation is made that requires another template instantiation to be made.

By default, the compiler uses a method called *implicit instantiation*, which is common practice, and results in having both the specification and definition available at point of instantiation. This involves placing template specifications in a header (for example, “.h”) file and the definitions in a source (for example, “.cpp”) file. Any file being compiled that includes a header file containing template specifications will instruct the compiler to implicitly include the corresponding “.cpp” file containing the definitions of the compiler.

For example, you may have the header file “tp.h”

```
template <typename A> void func(A var)
```

and source file “tp.cpp”

```
template <typename A> void func(A var)
{
    ...code...
}
```


Two files “file1.cpp” and “file2.cpp” that include “tp.h” will have file “tp.cpp” included implicitly to make the template definitions available to the compilation.

When generating dependencies, the compiler will only parse each implicitly included .cpp file once. This parsing avoids excessive compilation times in situations where a header file that implicitly includes a source file is included several times. If the .cpp file should be included implicitly more than once, the `-full-dependency-inclusion` switch ([on page 1-75](#)) can be used. (For example, the file may contain macro guarded sections of code.) This may result in more time required to generate dependencies.

If there is a desire not to use the implicit inclusion method then the `-no-implicit-inclusion` switch ([on page 1-76](#)) should be passed to the compiler. In the example, we have been discussing, “tp.cpp” will then be treated as a normal source file and should be explicitly linked into the final product.

Regardless of whether implicit instantiation is used or not, the compilation process involves compiling one or more source files and generating a “.ti” file corresponding to the source files being compiled. These “.ti” files are then used by the prelinker to determine the templates to be instantiated. The prelinker creates a “.ii” file and recompiles one or more of the files instantiating the required templates.

The prelinker ensures that only one instantiation of a particular template is generated across all objects. For example, the prelinker ensures that if both “file1.cpp” and “file2.cpp” invoked the template function with an int, that the resulting instantiation would be generated in just one of the objects.

Identifying Un-instantiated Templates

If for some reason the prelinker is unable to instantiate all the templates that are required for a particular link then a link error will occur. For example:

```
[Error li1021] The following symbols referenced in processor 'P0'
could not be resolved:
    'Complex<T1> Complex<T1>::_conjugate() const [with T1=short]
[_conjugate__16Complex__tm__2_sCFv_18Complex__tm__4_Z1Z]' refer-
enced from '.\Debug\main.doj'
    'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
[_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'
referenced from '.\Debug\main.doj'
    'T1 Complex<T1>::_getReal() const [with T1=short]
[_getReal__16Complex__tm__2_sCFv_Z1Z]' referenced from
'.\Debug\main.doj'
```

Linker finished with 1 error

Careful examination of the linker errors reveals which instantiations have not been made. Below are some examples.

Missing instantiation:

```
Complex<short> Complex<short>::_conjugate()
```

Linker Text:

```
'Complex<T1> Complex<T1>::_conjugate() const [with T1=short]
[_conjugate__16Complex__tm__2_sCFv_18Complex__tm__4_Z1Z]'
referenced from '.\Debug\main.doj'
```

Missing instantiation:

```
Complex<short> *Buffer<Complex<short>>::_getAddress()
```

Linker Text:

```
'T1 *Buffer<T1>::_getAddress() const [with T1=Complex<short>]
[_getAddress__33Buffer__tm__19_16Complex__tm__2_sCFv_PZ1Z]'
referenced from '.\Debug\main.doj'
```

Missing instantiation:

```
Short Complex<short>::_getReal()
```

Linker Text:

```
'T1 Complex<T1>::_getReal() const [with T1=short]
[_getReal__16Complex__tm__2_sCFv_ZlZ]' referenced from
'..\Debug\main.doj'
```

There could be many reasons for the prelinker being unable to instantiate these templates, but the most common is that the `.ti` and `.ii` files associated with an object file have been removed. Only source files that can contain instantiated templates will have associated `.ti` and `.ii` files, and without this information, the prelinker may not be able to complete its task. Removing the object file and recompiling will normally fix this problem.

Another possible reason for un-instantiated templates at link time is when implicit inclusion (described above) is disabled but the source code has been written to require it. Explicitly compiling the `.cpp` files that would normally have been implicitly included and adding them to the final link is normally all that is needed to fix this.

Another likely reason for seeing the linker errors above is invoking the linker directly. It is the compiler's responsibility to instantiate C++ templates, and this is done automatically if the final link is performed via the compiler driver. The linker itself contains no support for instantiating templates.

File Attributes

A file attribute is a name-value pair that is associated with a binary object, whether in an object file (`.doj`) or in a library file (`.dlb`). One attribute name can have multiple values associated with it. Attribute names and values are strings. A valid attribute name consists of one or more characters matching the following pattern:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

An attribute value is a non-empty character sequence containing any characters apart from NUL.

Attributes help with the placement of run-time library functions. All of the runtime library objects contain attributes which allow you to place time-critical library objects into internal (fast) memory. Using attribute filters in the LDF, you can place run-time library objects into internal or external (slow) memory, either individually or in groups.

This section describes:

- [“Automatically-Applied Attributes”](#)
- [“Default LDF Placement” on page 1-351](#)
- [“Sections versus Attributes” on page 1-352](#)
- [“Using Attributes” on page 1-353](#)

Automatically-Applied Attributes

By default, the compiler automatically applies a number of attributes when compiling a C/C++ file. [Figure 1-6](#) shows a content attribute tree.

For example, it applies the `Content` and `FuncName`, and `Encoding` attributes. These automatically-applied attributes can be disabled using the `-no-auto-attrs` switch ([on page 1-43](#)).

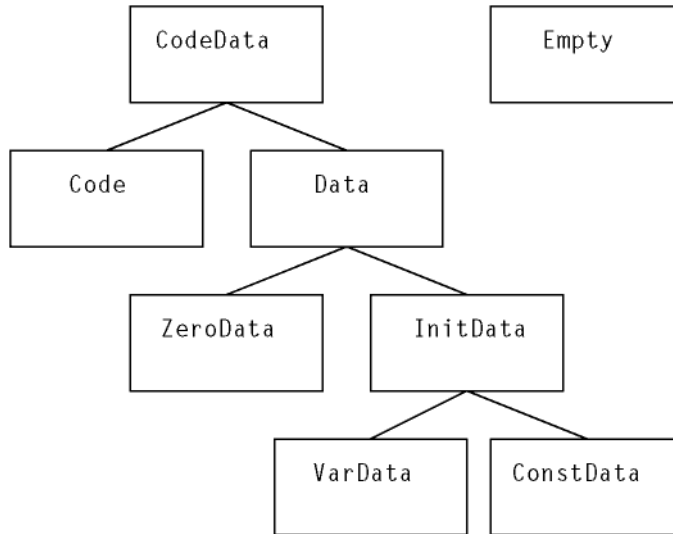


Figure 1-6. Content Attributes

Content Attributes

The `Content` attributes can be used to map binary objects according to their kind of content, as show by [Table 1-44](#).

Table 1-44. Values of the `Content` Attribute

Value	Description
<code>CodeData</code>	This is the most general value, indicating that the binary object contains a mix of content types.
<code>Code</code>	The binary object does not contain any global data, only executable code. This can be used to map binary objects into program memory, or into read-only memory.
<code>Data</code>	The binary object does not contain any executable code. The binary object may not be mapped into dedicated program memory. The kinds of data used in the binary object vary.

Table 1-44. Values of the `Content` Attribute (Cont'd)

Value	Description
<code>ZeroData</code>	The binary object contains only zero-initialized data. Its contents must be mapped into a memory section with the <code>ZERO_INIT</code> qualifier, to ensure correct initialization.
<code>InitData</code>	The binary object contains only initialized global data. The contents may not be mapped into a memory section that has the <code>ZERO_INIT</code> qualifier.
<code>VarData</code>	The binary object contains initialized variable data. It must be mapped into read-write memory, and may not be mapped into a memory section with the <code>ZERO_INIT</code> qualifier.
<code>ConstData</code>	The binary object contains only constant data (data declared with the C <code>const</code> qualifier). The data may be mapped into read-only memory (but see also the <code>-const-read-write</code> switch (on page 1-28) and its effects).
<code>Empty</code>	The binary object contains neither functions nor global data.

FuncName Attributes

The `FuncName` attributes are multi-valued attributes whose values are all the assembler linkage names of the defined names in `obj`.

Encoding Attributes

The `Encoding` attributes can be used to map binary objects according to the encoding of code they contain, as shown by [Table 1-45](#).

Table 1-45. Values of the `Encoding` Attribute

Value	Description
<code>SW</code>	The binary object contains only short-word code (2146x processors only).
<code>NW</code>	The binary object contains only normal-word code.
<code>Mixed</code>	The binary object contains a mixture of short-word and normal-word code (2146x processors only).

Default LDF Placement

The default `.ldf` file is written so that the order of preference for putting an object in section `seg_dmda` or `seg_pmco` depends on the value of the `prefersMem` attribute. Precedence is given in the following order:

1. Highest priority is given to binary objects that have a `prefersMem` attribute with a value of `internal`.
2. Next priority is given to binary objects that have no `prefersMem` attribute, or a `prefersMem` attribute with a value that is neither `internal` nor `external`.
3. Lowest priority is given to binary objects with a `prefersMem` attribute with the value `external`.

Although the default `.ldf` files only reference the values `internal` and `external`, `prefersMem` may have other values. For example, an object using a value such as `L2` will be given second priority, as the value is neither `internal` nor `external`. You may modify your `.ldf` file to assign appropriate priority to any value you choose, by mapping objects with higher-priority before objects with lower-priority values.

The `prefersMemNum` attribute is similar to the `prefersMem` attribute, but is given numerical values instead of textual values. This makes it easier to assign priority when there are many different levels, because you can use relational comparisons in the `.ldf` file instead of just equalities and inequalities. [Table 1-46](#) shows the numerical values used by the run-time library for each corresponding `prefersMem` attribute value.

Table 1-46. Values for `prefersMemNum` attribute

prefersMem attribute value	prefersMemNum attribute value
<code>internal</code>	30
<code>any</code>	50
<code>external</code>	70

Sections versus Attributes

File attributes and section qualifiers ([on page 1-348](#)) can be thought of as being somewhat similar, since they can both affect how the application is linked. There are important differences, however. These differences will affect whether you choose to use sections or file attributes to control the placement of code and data.

Granularity

Individual components – global variables and functions – in a binary object can be assigned different sections, then those section assignments can be used to map each component of the binary object differently. In contrast, an attribute applies to the whole binary object. This means you do not have as fine control over individual components using attributes as when using sections.

“Hard” Versus “Soft”

A section qualifier is a *hard* constraint: when the linker maps the object file into memory, it must obey all the section qualifiers in the object file, according to instructions in the .LDF file. If this cannot be done, or if the .LDF file does not give sufficient information to map a section from the object file, the linker will report an error.

With attributes, the mapping is *soft*: the default LDFs use the `prefersMem` attribute as a guide to give a better mapping in memory, but if this cannot be done, the linker will not report an error. For example, if there are more objects with `prefersMem=internal` than will fit into internal memory, the remaining objects will spill over into external memory. Likewise, if there are less objects with the attribute `prefersMem!=external` than are needed to fill internal memory, some objects with the `prefersMem=external` attribute may get mapped to internal memory.

Section qualifiers are rules that must be obeyed, while attributes are guidelines, defined by convention, that can be used if convenient and ignored if inconvenient. The `Content` attribute is an example: you can use the `Content` attribute to map `Code` and `ConstData` binary objects into read-only memory, if this is a convenient partitioning of your application. However, you need not do so if you choose to map your application differently.

Number of Values

Any given element of an object file is assigned exactly one section qualifier, to determine into which section it should be mapped. In contrast, an object file may have many attributes (or even none), and each attribute may have many different values. Since attributes are optional and act as guidelines, you need only pay attention to the attributes that are relevant to your application.

Using Attributes

You can add attributes to a file in two ways:

- Use `#pragma file_attr` ([on page 1-206](#)).
- Use the `-file-attr` switch ([on page 1-33](#)).

Refer to “[Example 1](#)” and “[Example 2](#)” on the use of attributes.

The run-time libraries have attributes associated with the objects in them. For more information on the attributes in run-time library objects, see “Library Attributes” in the *VisualDSP++ 5.0 Run-Time Library Manual for SHARC Processors*.

Example 1

This example demonstrates how to use attributes to encourage the placement of library functions in internal memory.

Suppose the file `test.c` exists, as shown below:

```
#define MANY_ITERATIONS 500
void main(void) {
    int i;
    for (i = 0; i < MANY_ITERATIONS; i++) {
        fft_lib_function();
        frequently_called_lib_function();
    }
    rarely_called_lib_function();
}
```

Also suppose:

- The objects containing `frequently_called_lib_function` and `rarely_called_lib_function` are both in the standard library, and have the attribute `prefersMem=any`.
- There is only enough internal memory to map `fft_lib_function` (which has `prefersMem=internal`) and one other library function into internal memory.
- The linker chooses to map `rarely_called_lib_function` to internal memory.

For optimal performance in this example, `frequently_called_lib_function` should be mapped to the internal memory in preference to `rarely_called_lib_function`.

The `.ldf` file defines the following macro `$OBS_LIBS_INTERNAL` to store all the objects that the linker should try to map to internal memory:

```
$OBS_LIBS_INTERNAL =
    $OBJECTS{prefersMem("internal")},
    $LIBRARIES{prefersMem("internal")};
```

If they do not all fit in internal memory, the remainder get placed in external memory – no linker error will occur. You must add the object that contains `frequently_called_lib_function` to this macro. Add a line to the LDF after the initial setting of this variable:

```
$OBS_LIBS_INTERNAL =
    $OBS_LIBS_INTERNAL
    $OBJECTS{ libFunc("frequently_called_lib_function") };
```

This ensures that the binary object that defines `frequently_called_lib_function` is among those to which the linker gives highest priority when mapping binary objects to internal memory.

Note that it is not necessary for you to know which binary object defines `frequently_called_lib_function` (or even which library). The binary objects in the run-time libraries all define the `libFunc` attribute so that you can select the binary objects for particular functions without needing to know exactly where in the libraries a function is defined.

The modified line uses this attribute to select the binary object(s) for `frequently_called_lib_function` and append them to the `$OBS_LIBS_INTERNAL` macro. The `.ldf` file maps objects in `$OBS_LIBS_INTERNAL` to internal memory in preference to other objects. Therefore, `frequently_called_lib_function` gets mapped to L1.

Example 2

Suppose you want the contents of `test.c` to get mapped to external memory by preference. You can do this by adding the following pragma to the top of `test.c`:

```
#pragma file_attr("prefersMem=external")
```

or use the `-file-attr` switch on the following command line:

```
cc21k -file-attr prefersMem=external test.c
```

Both of these methods will mean that the resulting object file will have the attribute `prefersMem=external`. The `.ldf` files give objects with this attribute the lowest priority when mapping objects into internal memory, so the object is less likely to consume valuable internal memory space which could be more usefully allocated to another function.



File attributes are used as guidelines rather than rules. If space is available in internal memory after higher-priority objects have been mapped, it is permissible for objects with `prefersMem=external` to be mapped into internal memory.

2 ACHIEVING OPTIMAL PERFORMANCE FROM C/C++ SOURCE CODE

This chapter provides guidance on tuning your application to achieve the best possible code from the compiler. Since implementation choices are available when coding an algorithm, understanding their impact is crucial to attaining optimal performance.

This chapter contains:

- [“General Guidelines” on page 2-3](#)
- [“Improving Conditional Code” on page 2-28](#)
- [“Loop Guidelines” on page 2-29](#)
- [“Using Built-In Functions in Code Optimization” on page 2-39](#)
- [“Smaller Applications: Optimizing for Code Size” on page 2-43](#)
- [“Using Pragmas for Optimization” on page 2-45](#)
- [“Useful Optimization Switches” on page 2-54](#)
- [“How Loop Optimization Works” on page 2-55](#)
- [“Assembly Optimizer Annotations” on page 2-80](#)

This chapter helps you get maximal code performance from the compiler. Most of these guidelines also apply when optimizing for minimum code size, although some techniques specific to that goal are also discussed.

The first section looks at some general principles, and explains how the compiler can help your optimization effort. Optimal coding styles are then considered in detail. Special features such as compiler switches, built-in functions, and pragmas are also discussed. The chapter ends with a short example to demonstrate how the optimizer works.

Small examples are included throughout this chapter to demonstrate points being made. Some show recommended coding styles, while others identify styles to be avoided or code that may be possible to improve. These are commented in the code as “GOOD” and “BAD” respectively.

General Guidelines

This section contains:

- [“How the Compiler Can Help” on page 2-4](#)
- [“Data Types” on page 2-15](#)
- [“Getting the Most From IPA” on page 2-17](#)
- [“Indexed Arrays Versus Pointers” on page 2-22](#)
- [“Using Function Inlining” on page 2-24](#)
- [“Using Inline asm Statements” on page 2-25](#)
- [“Memory Usage” on page 2-26](#)

Remember the following strategy when writing an application:

1. Choose the language as appropriate.
Your first decision is whether to implement your application in C/C++. Performance considerations may influence this decision. C++ code using only C features has very similar performance to pure C code. Many higher level C++ features (for example, those resolved at compilation, such as namespaces, overloaded functions and also inheritance) have no performance cost.

However, use of some other features may degrade performance. Carefully weigh performance loss against the richness of expression available in C++ (such as virtual functions or classes used to implement basic data types).
2. Choose an algorithm suited to the architecture being targeted. For example, the target architecture will influence any trade-off between memory usage and algorithm complexity.

General Guidelines

3. Code the algorithm in a simple, high-level generic form. Keep the target in mind, especially when choosing data types.
4. Tune critical code sections. After your application is complete, identify the most critical sections. Carefully consider the strengths of the target processor and make non-portable changes where necessary to improve performance.

How the Compiler Can Help

The compiler provides many facilities to help the programmer to achieve optimal performance, including the compiler optimizer, statistical profiler, Profile-Guided Optimizer (PGO), and interprocedural optimizers.

This section contains:

- [“Using the Compiler Optimizer” on page 2-4](#)
- [“Using Compiler Diagnostics” on page 2-5](#)
- [“Using the Statistical Profiler” on page 2-7](#)
- [“Using Profile-Guided Optimization” on page 2-8](#)
- [“Using Interprocedural Optimization” on page 2-13](#)

Using the Compiler Optimizer

There is a vast difference in performance between code compiled optimized and code compiled non-optimized. In some cases, optimized code can run ten or twenty times faster. Always use optimization when measuring performance or shipping code as product.

The optimizer in the C/C++ compiler is designed to generate efficient code from source that has been written in a straightforward manner. The basic strategy for tuning a program is to present the algorithm in a way that gives the optimizer the best possible visibility of the operations and data, and hence the greatest freedom to safely manipulate the code. Future

releases of the compiler will continue to enhance the optimizer. Expressing algorithms simply will provide the best chance of benefiting from such enhancements.

Note that the default setting (or “debug” mode within the VisualDSP++ IDDE) is for non-optimized compilation in order to assist programmers in diagnosing problems with their initial coding. The optimizer is enabled in VisualDSP++ by checking the **Enable optimization** checkbox under the **Project Options ->Compile** tab or by using the `-O` switch ([on page 1-50](#)). A “release” build from within VisualDSP++ automatically enables optimization.

Using Compiler Diagnostics

There are many features of the C and C++ languages that, while legal, often indicate programming errors. There are also aspects that are valid but may be relatively expensive for an embedded environment. The compiler can provide the following diagnostics, which may save time and effort in characterizing source-related problems:

- Warnings and remarks
- Source and assembly annotations

These diagnostics are particularly important for obtaining high-performance code, since the optimizer aggressively transforms the application to get the best performance, discarding unused or redundant code; if this code is redundant because of a programming error (such as omitting an essential `volatile` qualifier from a declaration), then the code will behave differently from a non-optimized version. Using the compiler’s diagnostics may help you identify such situations before they become problems.

Warnings and Remarks

By default, the compiler emits warnings to the standard error stream at compile-time when it detects a problem with the source code. Warnings can be disabled individually, with the `-Wsuppress` switch ([on page 1-68](#))

General Guidelines

or as a class, with the `-w` switch ([on page 1-69](#)), disabling all warnings and remarks. However, disabling warnings is inadvisable until each instance has been investigated for problems.

A typical warning involves a variable being used before its value has been set.

Remarks are diagnostics that are less severe than warnings. Like warnings, they are produced at compile-time to the standard error stream, but unlike warnings, remarks are suppressed by default. Remarks are typically for situations that are probably correct, but less than ideal. Remarks may be enabled as a class with the `-wremarks` switch ([on page 1-69](#)) or the **Enable remarks** option.

A typical remark involves a variable being declared, but never used.

A remark may be promoted to a warning through the `-wwarn` switch ([on page 1-68](#)). Remarks and warnings may be promoted to an error through the `-werror` switch ([on page 1-68](#)). Here is a procedure for improving overall code quality:

1. Enable remarks and build the application. Gather all warnings and remarks generated.
2. Examine the generated diagnostics, and choose those message types that you consider most important. For example, you might select just `cc0223`, a remark that identifies implicitly-declared functions.
3. Promote those remarks and warnings to errors, using the `-werror` switch (for example, “`-werror 0223`”), and rebuild the application. The compiler will now fault such cases as errors, so you will have to fix the source to address the issues before your application will build.
4. Once your application rebuilds, repeat the process for the next most important diagnostics.

Achieving Optimal Performance from C/C++ Source Code

Diagnostics you might typically consider first include:

- cc0223: function declared implicitly
- cc0549: variable used before its value is set
- cc1665: variable is possibly used before its value is set, in a loop
- cc0187: use of “=” where “==” may have been intended
- cc1045: missing return statement at the end of non-void function
- cc0111: statement is unreachable

If you have particular cases that are correct for your application, do not let them prevent your application from building because you have raised the diagnostic to an error. For such cases, temporarily lower the severity again within the source file in question by using `#pragma diag` ([on page 1-160](#)).

Source and Assembly Annotations

By default, the compiler emits annotations that are embedded in the generated code—either in the object file or in the assembly source, depending on the output form you select. The source-related annotations can be viewed within the VisualDSP++ IDDE, while the assembly-related annotations give considerably more information about the intricacies of the generated code. Annotations can be used to find out why the compiler has generated code in a particular manner.

For more information, see “Assembly Optimizer Annotations” on [page 2-80](#).

Using the Statistical Profiler

Tuning an application begins with identifying the areas of the application are most frequently executed and therefore where improvements would provide the largest gains. The VisualDSP++ statistical profiler provides an easy way to find these areas. *VisualDSP++ 5.0 User's Guide* explains how to use the profiler in detail.

General Guidelines

The advantage of statistical profiling is that it is completely unobtrusive. Other forms of profiling insert instrumentation into the code, disturbing the original optimization, code size, and register allocation.

The best methodology is usually to compile with both optimization and debug information generation enabled. You can then obtain a profile of the optimized code while retaining function names and line number information. This gives you accurate results that correspond directly to the C/C++ source. Note that the compiler optimizer may have moved code between lines.

If you build your application optimized but without debug information generation, the profile will obtain statistics that relate directly to the assembly code. This kind of profile provides the most precise view of your application but not usually the easiest to use because you must relate assembly lines to the original source. Do not strip out function names when linking, since keeping function names means you can scroll through the assembly window to instructions of interest.

In complex code, you can locate the exact source lines by counting the loops, unless they are unrolled. Looking at the line numbers in the assembly file may also help. (Use the `-save-temps` switch to retain compiler generated assembly files, which will have the `.s` filename extension.) The compiler optimizer may have moved code around so that it does not appear in the same order as in your original source.

Using Profile-Guided Optimization

Profile-guided optimization (PGO) is an excellent way to tune the compiler's optimization strategy for the typical run-time behavior of a program. There are many program characteristics that cannot be known statically at compile-time but can be provided through PGO. The compiler can use this knowledge to improve its code generation. The benefits

include more accurate branch prediction, improved loop transformations, and reduced code size. The technique is most relevant where the behavior of the application over different data sets is expected to be very similar.

i PGO is supported in the simulator only.

Using Profile-Guided Optimization With a Simulator

The PGO process is illustrated in [Figure 2-1 on page 2-9](#).

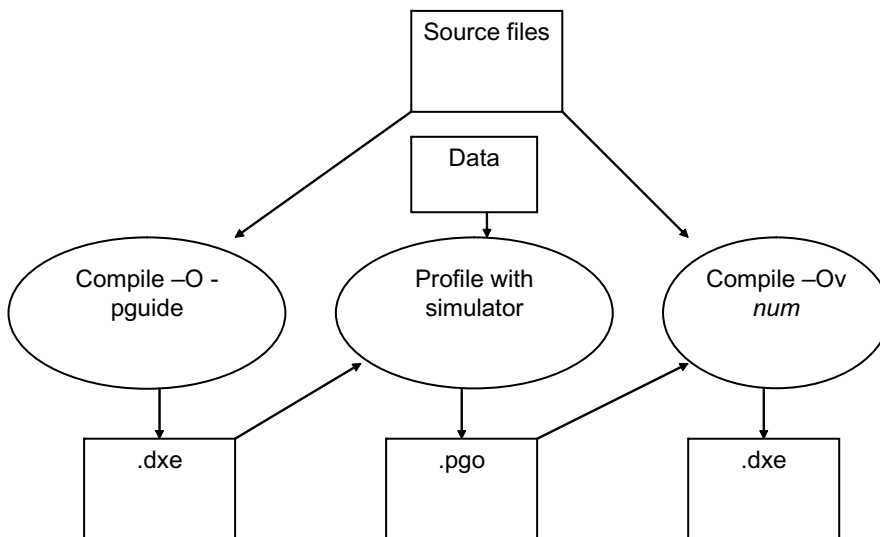


Figure 2-1. PGO Process

1. Compile the application is first compiled with the `-pguide` switch ([on page 1-57](#)) or **Prepare application to create new profile** option. This creates an executable file containing the necessary instrumentation for gathering profile data. For best results, use the **Enable optimization** option/`-O` switch ([on page 1-50](#)) or **Interprocedural analysis** option/`-ipa` ([on page 1-39](#)) switch.

General Guidelines

2. Gather the profile. Currently, this can only be done using a simulator. Run the executable with one or more training data sets. These training data sets should be representative of the data that you expect the application to process in the field. Note that unrepresentative training data sets can cause performance degradations when the application is used on real data. The profile is stored in a file with the extension `.pgo`.
3. Recompile the application using this gathered profile data. Place the `.pgo` file on the command line. Optimization should also be enabled at this stage.



When C/C++ source files are specified in a compiler command line, any `.pgo` files also specified will be used to guide their compilation. However, any recompilation due to `.doj` files provided on the command line will reread the same `.pgo` file as when the source was previously compiled. For example, `prof2.pgo` is ignored in the following commands:

```
cc21k -O f2.c -o f2.doj prof1.pgo
cc21k -o prog.dxe f1.asm f2.doj prof2.pgo
```

Using Profile-Guided Optimization With Non-Simulatable Applications

It may not be possible to run a complex application in its entirety in a simulation session (for example, if peripherals not modeled by the simulator are used). It may, however, still be possible to use PGO as follows.

1. If the application is structured in a modular fashion, it will be possible to extract the core performance-critical algorithm from the application.
2. Create a “wrapper” project, which can be run under simulation that drives input values into the core algorithm, replacing the portions of the application that can not be run under simulation. This project can be used to generate PGO information, which can sub-

sequently be used to optimize the full application. As described earlier, it is essential that the input values are representative of real data to achieve best performance.

3. Leave as much of the core algorithm unmodified as possible, keeping file and function names the same. The `.pgo` files generated from execution of the wrapper project can then be used to optimize the same functions in the full application by including the `.pgo` files in the full application build.



When compiling with a `.pgo` file, the compiler emits a warning and ignores the data for a function if it detects the function has changed from when the PGO data was generated. Therefore, any functions that you do modify to get the algorithm to work properly outside the application will not benefit from the profile information.

Profile-Guided Optimization and Multiple Source Uses

In some applications, it is convenient to build the same source file in more than one way within the same application. For example, a source file might be conditionally compiled with different macro settings. Alternatively, the same file might be compiled once, but linked more than once into the same application, in a multi-core or multi-processor environment. In such circumstances, the typical behaviors of each instance in the application might differ. You should identify the separate instances so that they can be profiled separately and optimized accordingly.

The `-pgo-session` switch ([on page 1-56](#)) (or **PGO session name** option) is used to separate profiles in such cases. It is used during both stage 1, where the compiler instruments the generated code for profiling, and during stage 3, where the compiler makes use of gathered profiles to guide the optimization.

General Guidelines

During stage 1, when the compiler instruments the generated code, if the `-pgo-session` switch is used, then the compiler marks the instrumentation as belonging to the session's `session-id`.

During stage 3, when the compiler reads gathered profiles, if the `-pgo-session` switch is used, then the compiler ignores all profile data not generated from code that was marked with the same `session-id`.

Therefore, the compiler can optimize each variant of the source's build according to how the variant is used, rather than according to an average of all uses.

Profile-Guided Optimization and the `-Ov` Switch

Note that when a `.pgo` file is placed on the command line, the `-O` optimization switch by default tries to balance between code performance and code-size considerations. It is equivalent to using the `-Ov 50` switch. To optimize solely for performance while using PGO, the switch `-Ov 100` should be used. The `-Ov n` switch ([on page 1-51](#)) is discussed further along with optimization for space in [“Smaller Applications: Optimizing for Code Size” on page 2-43](#).

Profile-Guided Optimization and Multiple PGO Data sets

When using profile-guided optimization with an executable constructed from multiple source files, using multiple PGO data sets will result in the creation of a temporary PGO information file (`.pgi` file). The file is used by the compiler and prelinker to ensure that temporary PGO files can be recreated and to identify cases where objects and PGO data sets are invalid.

The compiler reports an error if any of the PGO data files have been modified in between initial compilation of an object and any recompilation that occurs at the final link stage. To avoid this error, perform a full recompilation after running the application to generate `.pgo` data files.

When to Use Profile-Guided Optimization

PGO should always be performed as the last optimization step. If the application source code is changed after gathering profile data, this profile data becomes invalid. The compiler does not use profile data when it can detect that it is inaccurate. However, it is possible to change source code in a way that is not detectable to the compiler (for example, by changing constants). The programmer should ensure that the profile data used for optimization remains accurate.

For more details on PGO, refer to [“Optimization Control” on page 1-82](#).

Using Interprocedural Optimization

To obtain the best performance, the optimizer often requires information that can only be determined by looking outside the function that it is optimizing. For example, it helps to know what data can be referenced by pointer parameters or if a variable actually has a constant value. The `-ipa` compiler switch ([on page 1-39](#)) enables interprocedural analysis (IPA), which can make this information available. When this switch is used, the compiler is called again from the link phase to recompile the program using additional information obtained during previous compilations.

This gathered information is stored within the object file generated during initial compilation. IPA retrieves the gathered information from the object file during linking, and uses it to recompile available source files where beneficial. Because recompilation is necessary, IPA-built modules in libraries can contribute to the optimization of application sources, but do not themselves benefit from IPA, as their source is not available for recompilation.

Because it only operates at link time, the effects of IPA are not seen if you compile with the `-S` switch ([on page 1-61](#)). To see the assembly file when IPA is enabled, use the `-save-temps` switch ([on page 1-61](#)), and look at the `.s` file produced after your program has been built.

General Guidelines

As an alternative to IPA, you can achieve many of the same benefits by adding pragma directives and other declarations such as `__builtin_aligned` to provide information to the compiler about how each function interacts with the rest of the program.

These directives are further described in [“Using `__builtin_aligned`” on page 2-19](#) and [“Using Pragmas for Optimization” on page 2-45](#).

Data Types

[Table 2-1](#) shows the following scalar data types that the compiler supports.

Table 2-1. Scalar Data Types

Single-Word Fixed-Point Data Types: Native Arithmetic	
char	32-bit signed integer
unsigned char	32-bit unsigned integer
short	32-bit signed integer
unsigned short	32-bit unsigned integer
int	32-bit signed integer
unsigned int	32-bit unsigned integer
long	32-bit signed integer
unsigned long	32-bit unsigned integer
Floating-Point Data Types: Native Arithmetic	
float	32-bit floating point Note: Default when the Double size option is set to 32 bits, or the <code>-double-size-32</code> switch is used.
double	32-bit floating point
Floating-Point Data Types: Emulated Arithmetic	
double	64-bit floating-point Note: Default when the Double size option is set to 64 bits, or the <code>-double-size-64</code> switch is used.
long double	64-bit floating-point

Fractional data types are represented using the integer types. Manipulation of these is best done using of built-in functions, described in [“Using System Support Built-In Functions” on page 2-39](#).

Avoiding Emulated Arithmetic

Arithmetic operations for some data types are implemented by library functions because the processor hardware does not directly support these types. Consequently, operations for these types are far slower than native operations (sometimes by a factor of a hundred) and also produce larger code. These types are marked as “Emulated Arithmetic” in [“Data Types” on page 2-15](#).

The hardware does not provide direct support for division, so division and modulus operations are almost always multi-cycle operations, even on integral type inputs. If the compiler has to issue a full-division operation, it usually needs to call a library function. One instance in which a library call is avoided is for integer division when the divisor is a compile-time constant and is a power of two. In that case, the compiler generates a shift instruction. Even then, a few fix-up instructions are needed after the shift if the types are signed. If you have a signed division by a power of two, consider whether you can change it to unsigned in order to obtain a single-instruction operation.

When the compiler has to generate a call to a library function for one of the arithmetic operators that are not supported by the hardware, performance suffers not only because the operation takes multiple cycles, but also because the effectiveness of the compiler optimizer is reduced.

For example, such operations in a loop can prevent the compiler from using efficient zero-overhead hardware loop instructions. Also, calling the library to perform the required operation can change values held in scratch registers before the call, so the compiler has to generate more stores and loads from the data stack to keep values required after the call returns. Emulated arithmetic operators should therefore be avoided where possible, especially in loops.

Getting the Most From IPA

Interprocedural analysis (IPA) is designed to try to propagate information about the program to parts of the optimizer that can use it. This section looks at what information is useful, and how to structure your code to make this information easily accessible for analysis.

The performance features are:

- [“Initialize Constants Statically”](#)
- [“Dual Word-Aligning Your Data” on page 2-18](#)
- [“Using __builtin_aligned” on page 2-19](#)
- [“Avoiding Aliases” on page 2-21](#)

Initialize Constants Statically

IPA identifies variables that have only one value and replaces them with constants, resulting in a host of benefits for the optimizer’s analysis. For this to happen a variable must have a single value throughout the program. If the variable is statically initialized to zero, as all global variables are by default, and is subsequently assigned some other value at another point in the program, then the analysis sees two values and does not consider the variable to have a constant value.

For example,

```
// BAD: IPA cannot see that val is a constant
#include <stdio.h>
int val; // initialized to zero

void init() {
    val = 3; // re-assigned
}

void func() {
    printf("val %d",val);
}
```

General Guidelines

```
}  
  
int main() {  
    init();  
    func();  
}
```

The code is better written as

```
// GOOD: IPA knows val is 3.  
#include <stdio.h>  
const int val = 3; // initialized once  
  
void init() {  
}  
  
void func() {  
    printf("val %d",val);  
}  
  
int main() {  
    init();  
    func();  
}
```

Dual Word-Aligning Your Data

This section applies to the dual compute-block architecture found in the ADSP-2116x, ADSP-2126x and ADSP-2136x processors.

To make most efficient use of the hardware, it must be kept fed with data. In many algorithms, the balance of data accesses to computations is such that, to keep the hardware fully utilized, data must be fetched with loads wider than 32 bits.

For external data, the ADSP-2116x chips require that dual-word memory accesses reference dual-word-aligned addresses. Therefore, for the most efficient code generation, ensure that your data buffers are dual-word-aligned.

The compiler helps to establish the alignment of array data. Top-level arrays are allocated at dual-word-aligned addresses, regardless of their data types. In order to do this for local arrays, the compiler also ensures that stack frames are kept dual-word-aligned. However, arrays within structures are not aligned beyond the required alignment for their type. It may be worth using the `#pragma align 2` directive to force the alignment of arrays in this case.

If you write programs that pass only the address of the first element of an array as a parameter, and loop that process through these input arrays, an element at a time (starting at element zero), then IPA should be able to establish that the alignment is suitable for full-width accesses.

Where an inner loop processes a single row of a multi-dimensional array, try to ensure that each row begins on a two-word boundary. In particular, two-dimensional arrays should be defined in a single block of memory rather than as an array of pointers to rows all separately allocated with `malloc`. It is difficult for the compiler to keep track of the alignment of the pointers in the latter case. It may also be necessary to insert dummy data at the end of each row to make the row length a multiple of two words.

Using `__builtin_aligned`

This section applies to the dual compute-block architecture found in the ADSP-2116x, ADSP-2126x and ADSP-2136x processors.

To avoid the need to use IPA to propagate alignment, and for situations when IPA cannot guarantee the alignment (but you can), use the `__builtin_aligned` function to assert the alignment of important pointers, meaning that the pointer points to data that is aligned. Remember when adding this declaration that you are responsible for making sure it is valid, and that if the assertion is not true, the code produced by the compiler is likely to malfunction.

General Guidelines

The assertion is particularly useful for function parameters, although you may assert that any pointer is aligned. For example, when compiling the function:

```
// BAD: without IPA, compiler doesn't know the alignment of a and b.
void copy(char *a, char *b) {
    int i;
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

the compiler does not know the alignment of pointers `a` and `b` if IPA is not being used. However, by modifying the function to:

```
// GOOD: both pointer parameters are known to be aligned.
void copy(char *a, char *b) {
    int i;
    __builtin_aligned(a, 4);
    __builtin_aligned(b, 4);
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

the compiler can be told that the pointers are aligned on dual-word boundaries. To assert instead that both pointers are always aligned one char before a dual-word boundary, use:

```
// GOOD: both pointer parameters are known to be misaligned.
void copy(char *a, char *b) {
    int i;
    __builtin_aligned(a+1, 4);
    __builtin_aligned(b+1, 4);
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

The expression used as the first parameter to the built-in function obeys the usual C rules for pointer arithmetic. The second parameter should give the alignment in words as a literal constant.

Avoiding Aliases

It may seem that the iterations can be performed in any order in the following loop:

```
// BAD: a and b may alias each other.
void fn(char a[], char b[], int n) {
    int i;
    for (i = 0; i < n; ++i)
        a[i] = b[i];
}
```

but `a` and `b` are both parameters, and, although they are declared with `[]`, they are pointers that may point to the same array. When the same data may be reachable through two pointers, they are said to alias each other.

If IPA is enabled, the compiler looks at the call sites of `fn` and tries to determine whether `a` and `b` can ever point to the same array.

Even with IPA, it is easy to create what appears to the compiler as an alias. The analysis works by associating pointers with sets of variables that they may refer to some point in the program. If the sets for two pointers intersect, then both pointers are assumed to point to the union of the two sets.

If `fn` above were called only in two places, with global arrays as arguments, then IPA would have the results shown below:

```
// GOOD: sets for a and b do not intersect: a and b are not aliases.
fn(glob1, glob2, N);
fn(glob1, glob2, N);

// GOOD: sets for a and b do not intersect: a and b are not aliases.
fn(glob1, glob2, N);
fn(glob3, glob4, N);

// BAD: sets intersect - both a and b may access glob1;
// a and b may be aliases.
fn(glob1, glob2, N);
fn(glob3, glob1, N);
```

General Guidelines

The third case arises because IPA considers the union of all calls at once, rather than considering each call individually, when determining whether there is a risk of aliasing. If each call were considered individually, IPA would have to take flow control into account and the number of permutations would significantly lengthen compilation time.

The lack of control flow analysis can also create problems when a single pointer is used in multiple contexts. For example, it is better to write

```
// GOOD: p and q do not alias.
int *p = a;
int *q = b;
    // some use of p
    // some use of q
```

than

```
// BAD: uses of p in different contexts may alias.
int *p = a;
    // some use of p
p = b;
    // some use of p
```

because the latter may cause extra apparent aliases between the two uses.

Indexed Arrays Versus Pointers

The C language allows a program to access data from an array in two ways: either by indexing from an invariant base pointer, or by incrementing a pointer. The following two versions of vector addition illustrate the two styles:

Style 1: using indexed arrays (indexing from a base pointer)

```
void va_ind(const short a[], const short b[], short out[], int n) {
    int i;
    for (i = 0; i < n; ++i)
        out[i] = a[i] + b[i];
}
```

Style 2: incrementing a pointer

```
void va_ptr(const short a[], const short b[], short out[], int n) {  
    int i;  
    short *pout = out;  
    const short *pa = a, *pb = b;  
    for (i = 0; i < n; ++i)  
        *pout++ = *pa++ + *pb++;  
}
```

Trying Pointer and Indexed Styles

One might hope that the chosen style would not make any difference to the generated code, but this is not always the case. Sometimes, one version of an algorithm generates better optimized code than the other, but it is not always the same style that is better.



Try both pointer and index styles.

The pointer style introduces additional variables that compete with the surrounding code for resources during the compiler optimizer's analysis. Array accesses, on the other hand, must be transformed to pointers by the compiler, and sometimes this is accomplished better by hand.

The best strategy is to start with array notation. If the generated code looks unsatisfactory, try using pointers. Outside the critical loops, use the indexed style, since it is easier to understand.

Using Function Inlining

Function inlining may be used in two ways:

- By annotating functions in the source code with the `inline` keyword. In this case, function inlining is performed only when optimization is enabled.
- By turning on automatic inlining with the `-Oa` switch ([on page 1-51](#)) or the **Inlining -> Automatic** option, automatically enabling optimization.



Inline small, frequently executed functions.

You can use the compiler's `inline` keyword to indicate that functions should have code generated inline at the point of call. Doing this avoids various costs such as program flow latencies, function entry and exit instructions and parameter passing overheads.

Using an `inline` function also has the advantage that the compiler can optimize through the inline code and does not have to assume that scratch registers and condition states are modified by the call. Prime candidates for inlining are small, frequently-used functions because they cause the least code-size increase while giving most performance benefit.

As an example of the usage of the `inline` keyword, the function below sums two input parameters and returns the result.


```
// GOOD: use of the inline keyword.  
inline int add(int a, int b) {  
    return (a+b);  
}
```

Inlining has a code-size-to-performance trade-off that should be considered. With `-Oa`, the compiler automatically inlines small functions where possible. If the application has a tight upper code-size limit, the resulting code-size expansion may be too great. Consider using automatic inlining in conjunction with the `-Ov num` switch ([on page 1-51](#)) or the **Optimize**

for code speed/size slider to restrict inlining (and other optimizations with a code-size cost) to parts of the application that are performance-critical. It is discussed in more detail later in this chapter.

Using Inline `asm` Statements

The compiler allows use of inline `asm` statements to insert small sections of assembly into C code.

 Avoid the use of inline `asm` statements where built-in functions may be used instead.

The compiler does not intensively optimize code that contains inline `asm` statements because it has little understanding about what the code in the statement does. In particular, use of an `asm` statement in a loop may inhibit useful transformations.


The compiler has a large number of built-in functions that generate specific hardware instructions. These are designed to allow the programmer to more finely tune the code produced by the compiler, or to allow access to system support functions. A complete list of compiler's built-in functions is given in [“Compiler Built-In Functions” on page 1-148](#).

Use of these built-in functions is much preferred to using inline `asm` statements. Since the compiler knows what each built-in function does, it can easily optimize around them. Conversely, since the compiler does not parse `asm` statements, it does not know what they do, and so is hindered in optimizing code that uses them. Note also that errors in the text string of an `asm` statement are caught by the assembler and not by the compiler.

Examples of efficient use of built-in functions are given in [“Using System Support Built-In Functions” on page 2-39](#).

Memory Usage

The compiler, in conjunction with the use of the linker description file (.ldf), allows the programmer control over where data is placed in memory. This section describes how to best lay out data for maximum performance.

 Try to put arrays into different memory sections.

The processor hardware can support two memory operations on a single instruction line, combined with a compute instruction. However, two memory operations complete in one cycle only if the two addresses are situated in different memory blocks. If both access the same block, the processor stalls.

Consider the dot product loop below. Because data is loaded from both array a and array b in every iteration of the loop, it may be useful to ensure that these arrays are located in different blocks.

```
// BAD: compiler assumes that two memory accesses together may
give a stall.
for (i=0; i<100; i++)
    sum += a[i] * b[i];
```

The “Dual Memory Support Language Keywords” compiler extension (see [“Dual Memory Support Keywords \(pm dm\)” on page 1-132](#)) can improve the compiler’s use of the memory system. Placing a pm qualifier before the type definition tells the compiler that the array is located in what is referenced as “Program Memory” (pm).

The memory of the SHARC processor is in one unified address space (except for the ADSP-21020 processor) and there is no restriction on where in memory program code or data can be placed. However, the default .ldf files ensure that pm-qualified data is placed in a different memory block than non-qualified (or dm-qualified) data, thus allowing two accesses to occur simultaneously without incurring a stall. The mem-

ory block used for pm-qualified data in the default .ldf files is the same memory block as is used for the program code, hence the name “Program Memory”.

To allow simultaneous accesses to the two buffers, modify the array declaration of either a or b program by adding the pm qualifier. Also add the pm qualifier to the declarations of any pointers that point to the pm buffer.

For example,

```
pm int a[100];
```

and any pointers to the buffer a become, for example,

```
pm int *p = a;
```

Note that only global or static data can be explicitly placed in Program Memory.

Improving Conditional Code

When compiling conditional statements, the compiler attempts to predict whether the condition will usually evaluate to true or to false, and will arrange for the most efficient path of execution to be that which is expected to be most commonly executed.

You can use the `expected_true` and `expected_false` built-in functions to control the compiler's behavior for specific cases. By using these functions, you can tell the compiler which way a condition is most likely to evaluate, and so influence the default flow of execution. For example,

```
if (buffer_valid(data_buffer))  
    if (send_msg(data_buffer))  
        system_failure();
```

shows two nested conditional statements. If it was known that `buffer_valid()` would usually return true, but that `send_msg()` would rarely do so, the code could be written as

```
if (expected_true(buffer_valid(data_buffer)))  
    if (expected_false(send_msg(data_buffer)))  
        system_failure();
```

See [“Compiler Performance Built-In Functions” on page 1-154](#) (on `expected_true` and `expected_false` functions) for more information.

The compiler can also determine the most commonly-executed branches automatically, using profile-guided optimization. See [“Optimization Control” on page 1-82](#) for more details.

Loop Guidelines

Loops are where an application ordinarily spends the majority of its time. It is therefore useful to look in detail at how to help the compiler to produce the most efficient code.

This section describes:

- [“Keeping Loops Short” on page 2-30](#)
- [“Avoiding Unrolling Loops” on page 2-30](#)
- [“Avoiding Loop-Carried Dependencies” on page 2-31](#)
- [“Avoiding Loop Rotation by Hand” on page 2-32](#)
- [“Avoiding Array Writes in Loops” on page 2-33](#)
- [“Inner Loops vs. Outer Loops” on page 2-33](#)
- [“Avoiding Conditional Code in Loops” on page 2-34](#)
- [“Avoiding Placing Function Calls in Loops” on page 2-35](#)
- [“Avoiding Non-Unit Strides” on page 2-35](#)
- [“Loop Control” on page 2-36](#)
- [“Using the Restrict Qualifier” on page 2-37](#)
- [“Avoiding Long Latencies” on page 2-38](#)

Keeping Loops Short

For best code efficiency, loops should be short. Large loop bodies are usually more complex and difficult to optimize. Large loops may also require register data to be stored in memory, which decreases code density and execution performance.

Avoiding Unrolling Loops

Do not unroll loops yourself. Not only does loop unrolling make the program harder to read but it also prevents optimization by complicating the code for the compiler.

```
// GOOD: the compiler unroll if it helps.
void va1(const short a[], const short b[], short c[], int n) {
    int i;
    for (i = 0; i < n; ++i) {
        c[i] = b[i] + a[i];
    }
}

// BAD: harder for the compiler to optimize.
void va2(const short a[], const short b[], short c[], int n) {
    short xa, xb, xc, ya, yb, yc;
    int i;
    for (i = 0; i < n; i+=2) {
        xb = b[i]; yb = b[i+1];
        xa = a[i]; ya = a[i+1];
        xc = xa + xb; yc = ya + yb;
        c[i] = xc; c[i+1] = yc;
    }
}
```

Avoiding Loop-Carried Dependencies

A loop-carried dependency exists when a computation in a given iteration of a loop cannot be completed without knowledge of values calculated in earlier iterations. When a loop has such dependencies, the compiler cannot overlap loop iterations. Some dependencies are caused by scalar variables that are used before they are defined in a single iteration.

However, if the loop-carried dependency is part of a *reduction* computation, the optimizer can reorder iterations. Reductions are loop computations that reduce a vector of values to a scalar value using an associative and commutative operator. A multiply and accumulate in a loop is a common example of a reduction.

```
// BAD: loop-carried dependence in variable x.
for (i = 0; i < n; ++i)
    x = a[i] - x;

// GOOD: loop-carried dependence is a reduction.
for (i = 0; i < n; ++i)
    x += a[i] * b[i];
```

In the first case, the scalar dependency is the subtraction operation. The variable `x` is modified in a manner that would give different results if the iterations were performed out of order. In contrast, in the second case, because the addition operator is associative and commutative, the compiler can perform the iterations in any order and still get the same result. Other examples of reductions are bitwise and/or and min/max operators. The existence of loop-carried dependencies that are not reductions prevents the compiler from vectorizing a loop—that is, executing more than one iteration concurrently.

Floating-point addition is by default treated as associative and as a reduction operator. However, strictly speaking, rounding effects can change the result when the order of summation is varied. Use the `-no-fp-associative` compiler switch (on [page 1-46](#)) to ensure floating-point operations are executed in the same order as in the source code.

Avoiding Loop Rotation by Hand

Do not rotate loops by hand. Programmers are often tempted to “rotate” loops in DSP code by hand, attempting to execute loads and stores from earlier or future iterations at the same time as computation from the current iteration. This technique introduces loop-carried dependencies that prevent the compiler from rearranging the code effectively. It is better to give the compiler a simpler version, and leave the rotation to the compiler.

For example,

```
// GOOD: is rotated by the compiler.
int ss(short *a, short *b, int n) {
    int sum = 0;
    int i;
    for (i = 0; i < n; i++) {
        sum += a[i] + b[i];
    }
    return sum;
}

// BAD: rotated by hand—hard for the compiler to optimize.
int ss(short *a, short *b, int n) {
    short ta, tb;
    int sum = 0;
    int i = 0;
    ta = a[i]; tb = b[i];
    for (i = 1; i < n; i++) {
        sum += ta + tb;
        ta = a[i]; tb = b[i];
    }
    sum += ta + tb;
    return sum;
}
```

Rotating the loop required adding the scalar variables `ta` and `tb` and introducing loop-carried dependencies.

Avoiding Array Writes in Loops

Other dependencies can be caused by writes to array elements. In the following loop, the optimizer cannot determine whether the load from `a` reads a value defined on a previous iteration or one that is overwritten in a subsequent iteration.

```
// BAD: has array dependency.  
for (i = 0; i < n; ++i)  
    a[i] = b[i] * a[c[i]];
```

The optimizer can resolve access patterns where the addresses are expressions that vary by a fixed amount on each iteration. These are known as “induction variables”.

```
// GOOD: uses induction variables.  
for (i = 0; i < n; ++i)  
    a[i+4] = b[i] * a[i];
```

Inner Loops vs. Outer Loops

Inner loops should iterate more than outer loops.

The optimizer focuses on improving the performance of inner loops because this is where most programs spend the majority of their time. It is considered a good trade-off for an optimization to slow down the code before and after a loop to make the loop body run faster. Therefore, try to make sure that your algorithm also spends most of its time in the inner loop; otherwise it may actually run slower after optimization. If you have nested loops where the outer loop runs many times and the inner loop runs a small number of times, try to rewrite the loops so that the outer loop has fewer iterations.

Avoiding Conditional Code in Loops

If a loop contains conditional code, control-flow latencies may incur large penalties if the compiler has to generate conditional jumps within the loop. In some cases, the compiler is able to convert `if-then-else` and `?:` constructs into conditional instructions. In other cases, it can evaluate the expression entirely outside of the loop. However, for important loops, linear code should be written where possible.

There are several techniques for removing conditional code. For example, there is hardware support for `min` and `max`. The compiler usually succeeds in transforming conditional code equivalent to `min` or `max` into the single instruction. With particularly convoluted code the transformation may be missed, in which case it is better to use `min` or `max` in the source code.

The compiler can sometimes perform the loop transformation that interchanges conditional code and loop structures. Nevertheless, instead of writing

```
// BAD: loop contains conditional code.
for (i=0; i<100; i++) {
    if (mult_by_b)
        sum1 += a[i] * b[i];
    else
        sum1 += a[i] * c[i];
}
```

it is better to write

```
// GOOD: two simple loops can be optimized well.
if (mult_by_b) {
    for (i=0; i<100; i++)
        sum1 += a[i] * b[i];
} else {
    for (i=0; i<100; i++)
        sum1 += a[i] * c[i];
}
```

if this is an important loop.

Avoiding Placing Function Calls in Loops

The compiler usually is unable to generate a hardware loop if the loop contains a function call due to the expense of saving and restoring the context of a hardware loop. In addition to obvious function calls, such as `printf()`, can also prevent hardware loop generation operations, such as division, modulus, and some type coercions, that may implicitly call library functions. For more details, see [“Data Types” on page 2-15](#).

Avoiding Non-Unit Strides

If you write a loop, such as

```
// BAD: non-unit stride means division may be required.
for (i=0; i<n; i+=3) {
    // some code
}
```

then for the compiler to turn this into a hardware loop, it needs to work out the loop trip count. To do so, it must divide `n` by 3. The compiler may decide that this is worthwhile as it speeds up the loop, but division is an expensive operation. Try to avoid creating loop control variables with strides other than 1 or -1.

In addition, try to keep memory accesses in consecutive iterations of an inner loop contiguous. This is particularly applicable to multi-dimensional arrays.

Therefore,

```
// GOOD: memory accesses contiguous in inner loop
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        sum += a[i][j];
```

Loop Guidelines

is likely to be better than

```
// BAD: loop cannot be unrolled to use wide loads.
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        sum += a[j][i];
```

as the former is more amenable to vectorization.

Loop Control

Use `int` types for loop control variables and array indices. For loop control variables and array indices, it is always better to use signed `ints` rather than any other integral type. For other integral types, the C standard requires various type promotions and standard conversions that complicate the code for the compiler optimizer. Frequently, the compiler is still able to deal with such code and create hardware loops and pointer induction variables. However, it does make it more difficult for the compiler to optimize and may occasionally result in under-optimized code.

The same advice goes for using automatic (local) variables for loop control. Use automatic variables for loop control and loop exit test. It is easy for a compiler to see that an automatic scalar whose address is not taken may be held in a register during a loop. But it is not as easy when the variable is a global or a function static.

Therefore, code such as

```
// BAD: may need to reload globvar on every iteration.
for (i=0; i<globvar; i++)
    a[i] = a[i] + 1;
```

may not create a hardware loop if the compiler cannot be sure that the write into the array `a` does not change the value of the global variable. The `globvar` must be reloaded each time around the loop before performing the exit test.

In this circumstance, the programmer can make the compiler's job easier by writing:

```
// GOOD: easily becomes a hardware loop.
int upper_bound = globvar;
for (i=0; i<upper_bound; i++)
    a[i] = a[i] + 1;
```

Using the Restrict Qualifier

The `restrict` qualifier provides one way to help the compiler resolve pointer aliasing ambiguities. Accesses from distinct restricted pointers do not interfere with each other. The loads and stores in the following loop

```
// BAD: possible alias of arrays a and b
void copy(short *a, short *b) {
    int i;
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

may be disambiguated by writing

```
// GOOD: restrict qualifier tells compiler that memory
// accesses do not alias
void copy(short *a, const short *b) {
    int i;
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

The `restrict` keyword is particularly useful on function parameters. but it can be used on any variable declaration. For example, the `copy` function may also be written as:

```
void copy(short *a, short *b) {
    int * restrict p = a;
    for (i=0; i<100; i++)
```

Loop Guidelines

```
        a[i] = b[i];  
    }
```

Avoiding Long Latencies

All pipelined machines introduce stall cycles when you cannot execute the current instruction until a prior instruction has exited the pipeline. For example, the SHARC processor stalls for three cycles on a table lookup. `a[b[i]]` takes three cycles more than you would expect.

If a stall is seen empirically, but it is not obvious to you exactly why it is occurring, a good way to learn about the cause is the **Pipeline Viewer**. This can be accessed through **Debug Windows -> Pipeline Viewer** in the VisualDSP++ 5.0 IDDE. By single-stepping through the program, you can see where the stall occurs. Note that the **Pipeline Viewer** is only available within a simulator session.

Using Built-In Functions in Code Optimization

Built-in functions, also known as compiler intrinsics, provide a method for the programmer to efficiently use low-level features of the processor hardware while programming in C. Although this section does not cover all the built-in functions available, it presents some code examples where implementation choices are available to the programmer. For more information, refer to [“Compiler Built-In Functions” on page 1-148](#).

Using System Support Built-In Functions

Built-in functions are also provided to perform low-level system management, in particular for the manipulation of system registers (defined in `sysreg.h`). It is usually better to use these built-in functions rather than `inline asm` statements.

The built-in functions cause the compiler to generate efficient inline instructions and their use often results in better optimization of the surrounding code at the point where they are used. Using the built-in functions also usually results in improved code readability.

For more information on supported built-in functions, refer to [“Compiler Built-In Functions” on page 1-148](#).

Examples of the two styles are:

```
// BAD: uses inline asm statement
asm("#include <def21060.h>");
    // Bit definitions for the registers

void func_no_interrupts(void){
    // Check if interrupts are enabled.
    // If so, disable them, call the function, then re-enable.
    int enabled;
```

Using Built-In Functions in Code Optimization

```
asm("r0=0; bit tst MODE1 IRPTEN; if tf r0=r0+1; %0 = r0;"
    : "=d"(enabled) : : "r0");
if (enabled)
    asm("bit clr model IRPTEN;");          // Disable interrupts
func();                                  // Do something
if (enabled)
    asm("bit set model IRPTEN;");          // Re-enable interrupts
}

// GOOD: uses sysreg.h
#include <sysreg.h>                        // Sysreg functions
#include <def21060.h>                      // Bit definitions for the registers

void func_no_interrupts(void){
    // Check if interrupts are enabled.
    // If so, disable them, call the function, then re-enable.
    int enabled = sysreg_bit_tst(sysreg_MODE1, IRPTEN);
    if (enabled)
        sysreg_bit_clr(sysreg_MODE1, IRPTEN); // Disable interrupts
    func();                                  // Do something
    if (enabled)
        sysreg_bit_set(sysreg_MODE1, IRPTEN);
                                           // Re-enable interrupts
}
```

This example reads and returns the `CYCLES` register.

Using Circular Buffers

Circular buffers are useful in DSP-style code. They can be used in several ways. Consider the C code:

```
// GOOD: the compiler knows that b is accessed as a circular buffer
for (i=0; i<1000; i++) {
    sum += a[i] * b[i%20];
}
```

Achieving Optimal Performance from C/C++ Source Code

Clearly the access to array `b` is a circular buffer. When optimization is enabled, the compiler produces a hardware circular buffer instruction for this access.

Consider this more complex example.

```
// BAD: may not be able to use circular buffer to access b
for (i=0; i<1000; i+=n) {
    sum += a[i] * b[i%20];
}
```

In this case, the compiler does not know if `n` is positive and less than 20. If it is, then the access may be correctly implemented as a hardware circular buffer. On the other hand, if it is greater than 20, a circular buffer increment may not yield the same results as the C code.

The programmer has two options here.

The first option is to compile with the `-force-circbuf` switch. This tells the compiler that any access of the form `a[i%n]` should be considered as a circular buffer. Before using this switch, you should check that this assumption is valid for your application.

- The value of `i` must be positive.
- The value of `n` must be constant across the loop, and greater than zero (as the length of the buffer).
- The value of `a` must be a constant across the loop (as the base address of the circular buffer).
- The initial value of `i` must be such that `a[i]` refers a valid position within the circular buffer. This is because the circular buffer operations will take effect when advancing from position `a[i]` to either `a[i+m]` or `a[i-m]`, by addition or subtraction, respectively. If `a[i]`

is not initially valid, then any access before the first advancement will not access the buffer, and `a[i+m]` and `a[i-m]` will not be guaranteed to reference the buffer after advancement.



Circular buffer operations (which add or subtract the buffer length to a pointer) are semantically different from `a[i%n]` (which performs a modulo operation on an index, and then adds the result to a base pointer). If you use the `-force-circbuf` switch when the above conditions are not true, the compiler generates code that does not have the intended effect.

Second, and preferred, option, is to use built-in functions to perform the circular buffering. Two functions (`__builtin_circindex` and `__builtin_circptr`) are provided for this purpose.

To make it clear to the compiler that a circular buffer should be used, you may write either:

```
// GOOD: explicit use of circular buffer via __builtin_circindex
for (i=0, j=0; i<1000; i+=n) {
    sum += a[i] * b[j];
    j = __builtin_circindex(j, n, 20);
}
```

or

```
// GOOD: explicit use of circular buffer via __builtin_circptr
int *p = b;
for (i=0, j=0; i<1000; i+=n) {
    sum += a[i] * (*p);
    p = __builtin_circptr(p, n, b, 80);
}
```

For more information, refer to [“Compiler Built-In Functions” on page 1-148](#)).

Smaller Applications: Optimizing for Code Size

The same ethos for producing fast code also applies to producing small code. You should present the algorithm in a way that gives the optimizer clear visibility of the operations and data, and hence the greatest freedom to safely manipulate the code to produce small applications.

Once the program is presented in this way, the optimization strategy depends on the code-size constraint that the program must obey. The first step should be to optimize the application for full performance, using `-O` or `-ipa` switches. If this obeys the code-size constraints, then no more need be done.

The “optimize for space” switch `-Os` ([on page 1-51](#)), which may be used in conjunction with IPA, performs every performance-enhancing transformation except those that increase code size. In addition, the `-e` linker switch (`-flags-link -e` if used from the compiler command line) may be helpful (see [on page 1-33](#)). This operation performs section elimination in the linker to remove unneeded data and code. If the code produced with the `-Os` and `-flags-link -e` switches does not meet the code-size constraint, some analysis of the source code is required to try to reduce the code size further.

Note that loop transformations such as unrolling and software pipelining increase code size. But it is these loop transformations that also give the greatest performance benefit. Therefore, in many cases compiling for minimum code size produces significantly slower code than optimizing for speed.

The compiler provides a way to balance between the two extremes of `-O` and `-Os`. This is the sliding-scale `-Ov num` switch (adjustable using the optimization slider bar under **Project Options** in the VisualDSP++ IDDE), described [on page 1-51](#). The `num` parameter is a value between 0 and 100, where the lower value corresponds to minimum code size and

the upper to maximum performance. A value in-between is used to optimize the frequently-executed regions of code for maximum performance, while keeping the infrequently-executed parts as small as possible. The switch is most reliable when using profile-guided optimization (see [“Optimization Control” on page 1-82](#)) since the execution counts of the various code regions have been measured experimentally. Without PGO, the execution counts are estimated, based on the depth of loop nesting.



Avoid the use of inline code.

Avoid using the `inline` keyword to inline code for functions that are used a number of times, especially if they not very small. The `-Os` switch does not have any effect on the use of the `inline` keyword. It does, however, prevent automatic inlining (using the `-Oa` switch) from increasing the code size. Macro functions can also cause code expansion and should be used with care.

Using Pragmas for Optimization

Pragmas can assist optimization by allowing the programmer to make assertions or suggestions to the compiler. This section looks at how they can be used to finely tune source code. Refer to [“Pragmas” on page 1-158](#) for full details of how each pragma works; the emphasis here is in considering under what circumstances they are useful during the optimization process.

In most cases the pragmas serve to give the compiler information which it is unable to deduce for itself. It must be emphasized that the programmer is responsible for making sure that the information given by the pragma is valid in the context in which it is used. Use of a pragma to assert that a function or loop has a quality that it does not in fact have is likely to result in incorrect code and hence a malfunctioning application.

An advantage of the use of pragmas is that they allow code to remain portable, since they are normally ignored by a compiler that does not recognize them.

Function Pragmas

Function pragmas include `#pragma alloc`, `#pragma const`, `#pragma pure`, `#pragma result_alignment`, `#pragma regs_clobbered`, and `#pragma optimize_{off|for_speed|for_space|as_cmd_line}`.

#pragma alloc

This pragma asserts that the function behaves like the `malloc` library function. In particular, it returns a pointer to new memory that cannot alias any pre-existing buffers. In the following code,

```
// GOOD: uses #pragma alloc to disambiguate out from a and b
#pragma alloc
int *new_buf(void);
int *vmul(int *a, int *b) {
```

Using Pragmas for Optimization

```
int i;  
int *out = new_buf();  
for (i=0; i<100; i++)  
    out[i] = a[i] * b[i];  
}
```

the use of the pragma allows the compiler to be sure that the write into buffer `out` does not modify either of the two input buffers `a` or `b`, and therefore the iterations of the loop may be re-ordered.

#pragma const

This pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers), and the result returned is only a function of the parameter values. The pragma may be applied to a function prototype or definition. It helps the compiler since two calls to the function with identical parameters always yield the same result. In this way, calls to `#pragma const` functions may be hoisted out of loops if their parameters are loop independent.

#pragma pure

Like `#pragma const`, this pragma asserts to the compiler that a function does not have any side effects (such as modifying global variables or data buffers). However, the result returned may be a function of both the parameter values and any global variables. The pragma may be applied to a function prototype or definition. Two calls to the function with identical parameters always yield the same result provided that no global variables have been modified between the calls. Hence, calls to `#pragma pure` functions may be hoisted out of loops if their parameters are loop independent and no global variables are modified in the loop.

#pragma result_alignment

This pragma may be used on functions that have either pointer or integer results. When a function returns a pointer, the pragma is used to assert that the return result always has some specified alignment. Therefore, the above example might further be refined if it is known that the `new_buf` function always returns buffers which are aligned on a dual-word boundary.

```
// GOOD: uses pragma result_alignment to specify that out has
// strict alignment
#pragma alloc
#pragma result_alignment (2)
int *new_buf(void);

int *vmul(int *a, int *b) {
    int i;
    int *out = new_buf();
    for (i=0; i<100; i++)
        out[i] = a[i] * b[i];
}
```

Further details on this pragma may be found in [“#pragma result_alignment \(n\)” on page 1-186](#). Another more laborious way to achieve the same effect would be to use `__builtin_aligned` at every call site to assert the alignment of the returned result.

#pragma regs_clobbered

This pragma is a useful way to improve the performance of code that makes function calls. The best use of the pragma is to increase the number of call-preserved registers available across a function call. There are two complementary ways in which this may be done.

First of all, suppose that you have a function written in assembly that you wish to call from C source code. The `regs_clobbered` pragma may be applied to the function prototype to specify which registers are “clobbered” by the assembly function, that is, which registers may have

Using Pragmas for Optimization

different values before and after the function call. Consider for example a simple assembly function to add two integers and mask the result to fit into 8 bits:

```
_add_mask:
    modify(i7,-3);
    r2=255;
    r8=r8+r4;
    r0=r8 and r2;
    i12=dm(m7,i6);;
    jump(m14,i12)(DB); rframe; nop;
._add_mask.end
```

Clearly the function does not modify the majority of the scratch registers available and thus these could instead be used as call-preserved registers. In this way fewer spills to the stack would be needed in the caller function. Using the following prototype,

```
// GOOD: uses regs_clobbered to increase call-preserved register set.
#pragma regs_clobbered "r0, r2, i12, ASTAT"
int add_mask(int, int);
```

the compiler is told which registers are modified by a call to the `add_mask` function. The registers not specified by the pragma are assumed to preserve their values across such a call and the compiler may use these spare registers to its advantage when optimizing the call sites.

The pragma is also powerful when all of the source code is written in C. In the above example, a C implementation might be:

```
// BAD: function thought to clobber entire volatile register set
int add_mask(int a, int b) {
    return ((a+b)&255);
}
```

Since this function does not need many registers when compiled, it can be defined using:

```
// GOOD: function compiled to preserve most registers
#pragma regs_clobbered "r0, r2, i12, CCset"
int add_mask(int a, int b) {
    return ((a+b)&255);
}
```

to ensure that any other registers aside from `r0`, `r2`, `i12` and the condition codes are not modified by the function. If any other registers are used in the compilation of the function, they are saved and restored during the function prologue and epilogue.

In general, it is not very helpful to specify any of the condition codes as call-preserved as they are difficult to save and restore and are usually clobbered by any function. Moreover, it is usually of limited benefit to be able to keep them live across a function call. Therefore, it is better to use `CCset` (all condition codes) rather than `ASTAT` in the clobbered set above. For more information, refer to [“#pragma regs_clobbered string” on page 1-178](#).

#pragma optimize_{off|for_speed|for_space|as_cmd_line}

The `optimize_` pragma may be used to change the optimization setting on a function-by-function basis. In particular, it may be useful to optimize functions that are rarely called (for example, error handling code) for space (using `#pragma optimize_for_space`), whereas functions critical to performance should be compiled for maximum speed (using `#pragma optimize_for_speed`). The `#pragma optimize_off` is useful for debugging specific functions without increasing the size or decreasing the performance of the overall application unnecessarily.



The `#pragma optimize_as_cmd_line` resets the optimization settings to be those specified on the `cc21k` command line when the compiler was invoked. Refer to [“General Optimization Pragas” on page 1-173](#) for more information.

Loop Optimization Pragmas

Many pragmas are targeted towards helping to produce optimal code for inner loops. These are the `loop_count`, `no_vectorization`, `vector_for`, `all_aligned`, and `no_alias` pragmas.

#pragma loop_count

The `loop_count` pragma enables the programmer to inform the compiler about a loop's iteration count. The compiler is able to make more reliable decisions about the optimization strategy for a loop if it knows the iteration count range. If you know that the loop count is always a multiple of some constant, this can also be useful as it allows a loop to be partially unrolled or vectorized without the need for conditionally-executed iterations. Knowledge of the minimum trip count may allow the compiler to omit the guards that are usually required after software pipelining. (A “guard” is code generated by the compiler to test a condition at run-time rather than at compile-time.) Any of the parameters of the pragma that are unknown may be left blank.

An example of the use of the `loop_count` pragma might be:

```
// GOOD: the loop_count pragma gives compiler helpful information
// to assist optimization)
#pragma loop_count(/*minimum*/ 40, /*maximum*/ 100, /*modulo*/ 4)
for (i=0; i<n; i++)
    a[i] = b[i];
```

For more information, refer to “[#pragma loop_count \(min, max, modulo\)](#)” on page 1-169.

#pragma no_vectorization

Vectorization (executing more than one iteration of a loop in parallel) can slow down loops with very small iteration counts since a loop prologue and epilogue are required. The `no_vectorization` pragma can be used directly above a `for` or `do` loop to tell the compiler not to vectorize the loop.

#pragma vector_for

The `vector_for` pragma is used to help the compiler to resolve dependencies that would normally prevent it from vectorizing a loop. It tells the compiler that all iterations of the loop may be run in parallel with each other, subject to rearrangement of reduction expressions in the loop. In other words, there are no loop-carried dependencies except reductions. An optional parameter, `n`, may be given in parentheses to say that only `n` iterations of the loop may be run in parallel. The parameter must be a literal value. For example,

```
// BAD: cannot be vectorized due to possible alias between a and b
for (i=0; i<100; i++)
    a[i] = b[i] + a[i-4];
```

cannot be vectorized if the compiler cannot tell that the array `b` does not alias array `a`. But the pragma may be added to tell the compiler that in this case four iterations may be executed concurrently.

```
// GOOD: pragma vector_for disambiguates alias
#pragma vector_for (4)
for (i=0; i<100; i++)
    a[i] = b[i] + a[i-4];
```

Note that this pragma does not force the compiler to vectorize the loop. The optimizer checks various properties of the loop and does not vectorize it if it believes that it is unsafe or it is not possible to deduce information

Using Pragmas for Optimization

necessary to carry out the vectorization transformation. The pragma assures the compiler that there are no loop-carried dependencies, but there may be other properties of the loop that prevent vectorization.

In cases where vectorization is impossible, the information given in the assertion made by `vector_for` may still be put to good use in aiding other optimizations.

For more information, refer to “[#pragma vector_for](#)” on page 1-173.

#pragma SIMD_for

The `#pragma SIMD_for` is similar to the `vector_for` pragma but makes the weaker assertion that only two iterations may be issued in parallel. Further details are given in “[#pragma SIMD_for](#)” on page 1-168.

#pragma all_aligned

The `all_aligned` pragma is used as shorthand for multiple `__builtin_aligned` assertions. By prefixing a `for` loop with the pragma, it is asserted that every pointer variable in the loop is aligned on a word boundary at the beginning of the first iteration.

Therefore, adding the pragma to the following loop

```
// GOOD: uses all_aligned to inform compiler of alignment of a and b
#pragma all_aligned
for (i=0; i<100; i++)
    a[i] = b[i];
```

is equivalent to writing

```
// GOOD: uses __builtin_aligned to give alignment of a and b
__builtin_aligned(a, 4);
__builtin_aligned(b, 4);
for (i=0; i<100; i++)
    a[i] = b[i];
```


Achieving Optimal Performance from C/C++ Source Code

In addition, the `all_aligned` pragma may take an optional literal integer argument `n` in parentheses. This tells the compiler that all pointer variables are aligned on a word boundary at the beginning of the n^{th} iteration. Note that the iteration count begins at zero. Therefore,

```
// GOOD: uses all_aligned to inform compiler of alignment of a and b
#pragma all_aligned (3)
for (i=99; i>=0; i--)
    a[i] = b[i];
```

is equivalent to

```
// GOOD: uses __builtin_aligned to give alignment of a and b
__builtin_aligned(a+96, 4);
__builtin_aligned(b+96, 4);
for (i=99; i>=0; i--)
    a[i] = b[i];
```

For more information, refer to “[#pragma all_aligned](#)” on page 1-169 and “[Using __builtin_aligned](#)” on page 2-19.

#pragma no_alias

When immediately preceding a loop, the `no_alias` pragma asserts that no load or store in the loop accesses the same memory as any other. This helps to produce shorter loop kernels as it permits instructions in the loop to be rearranged more freely. See “[#pragma no_alias](#)” on page 1-172 for more information.

Useful Optimization Switches

Table 2-2 lists the compiler switches useful during the optimization process.

Table 2-2. C/C++ Compiler Optimization Switches

Switch Name	Description
<code>-const-read-write</code> on page 1-28	Specifies that data accessed via a pointer to <code>const</code> data may be modified elsewhere
<code>-flags-link -e</code> on page 1-33	Specifies linker section elimination
<code>-force-circbuf</code> on page 1-34	Treats array references of the form <code>array[i%n]</code> as circular buffer operations
<code>-ipa</code> on page 1-39	Turns on inter-procedural optimization. Implies use of <code>-O</code> . May be used in conjunction with <code>-Os</code> or <code>-Ov</code> .
<code>-no-fp-associative</code> on page 1-46	Does not treat floating-point multiply and addition as an associative
<code>-no-saturation</code> on page 1-48	Do not turn non-saturating operations into saturating ones
<code>-O</code> on page 1-50	Enables code optimizations and optimizes the file for speed
<code>-Os</code> on page 1-51	Optimizes the file for size
<code>-Ov num</code> on page 1-51	Controls speed vs. size optimizations (sliding scale)
<code>-pguide</code> on page 1-57	Adds instrumentation for the gathering of a profile as the first stage of performing profile-guided optimization
<code>-save-temps</code> on page 1-61	Saves intermediate files (for example, <code>.s</code>)

How Loop Optimization Works

Loop optimization is important to overall application performance, because any performance gain achieved within the body of a loop reaps a benefit for every iteration of that loop. This section provides an introduction to some of the concepts used in loop optimization, helping you to use the compiler features in this chapter.

This section contains:

- [“Terminology”](#)
- [“Loop Optimization Concepts” on page 2-58](#)
- [“A Worked Example” on page 2-77](#)

Terminology

This section describes terms that have particular meanings for compiler behavior.

Clobbered Register

A register is “clobbered” if its value is changed so that the compiler cannot usefully make assumptions about its new contents.

For example, when the compiler generates a call to an external function, the compiler considers all caller-preserved registers to be clobbered by the called function. Once the called function returns, the compiler cannot make any assumptions about the values of those registers. This is why they are called “caller-preserved.” If the caller needs the values in those registers, the caller must preserve them itself.

The set of registers clobbered by a function can be changed using `#pragma regs_clobbered`, and the set of registers changed by a `gnu asm` statement is determined by the clobber part of the `asm` statement.

How Loop Optimization Works

Live Register

A register is “live” if it contains a value needed by the compiler, and thus cannot be overwritten by a new assignment to that register. For example, to do “A = B + C”, the compiler might produce:

```
reg1 = load B           // reg1 becomes live
reg2 = load C           // reg2 becomes live
reg1 = reg1 + reg2      // reg2 ceases to be live;
                        // reg1 still live, but with a different
                        // value
store reg1 to A         // reg1 ceases to be live
```

Liveness determines which registers the compiler may use. In this example, since reg1 is used to load B, and that register must maintain its value until the addition, reg1 cannot also be used to load the value of C, unless the value in reg1 is first stored elsewhere.

Spill

When a compiler needs to store a value in a register, and all usable registers are already live, the compiler must store the value of one of the registers to temporary storage (the stack). This “spilling” process prevents the loss of a necessary value.

Scheduling

“Scheduling” is the process of re-ordering the program instructions to increase the efficiency of the generated code but without changing the program’s behavior. The compiler attempts to produce the most efficient schedule.

Loop Kernel

The “loop kernel” is the body of code that is executed once per iteration of the loop. It excludes any code required to set up the loop or to finalize it after completion.

Loop Prolog

A “loop prolog” is a sequence of code required to set the machine into a state whereby the loop kernel can execute. For example, the prolog may pre-load some values into registers ready for use in the loop kernel. Not all loops need a prolog.

Loop Epilog

A “loop epilog” is a sequence of code responsible for finalizing the execution of a loop. After each iteration of the loop kernel, the machine will be in a state where the next iteration can begin efficiently. The epilog moves values from the final iteration to where they need to be for the rest of the function to execute. For example, the epilog might save values to memory. Not all loops need an epilog.

Loop Invariant

A “loop invariant” is an expression that has the same value for all iterations of a loop. For example:

```
int i, n = 10;
for (i = 0; i < n; i++) {
    val += i;
}
```

The variable `n` is a loop invariant. Its value is not changed during the body of the loop, so `n` will have the value 10 for every iteration of the loop.

Hoisting

When the optimizer determines that some part of a loop is computing a value that is actually a loop invariant, it may move that computation to before the loop. This prevents the same value from being re-computed for every iteration. This is called “hoisting.”

Sinking

When the optimizer determines that some part of a loop is computing a value that is not used until the loop terminates, the compiler may move that computation to after the loop. This “sinking” process ensures the value is only computed using the values from the final iteration.

Loop Optimization Concepts

The compiler optimizer focuses considerable attention on program loops, as any gain in the loop’s performance reaps the benefits on every iteration of the loop. The applied transformations can produce code that appears to be substantially different from the structure of the original source code. This section provides an introduction to the compiler’s loop optimization, to help you understand why the code might be different.

This section describes:

- [“Software Pipelining” on page 2-59](#)
- [“Loop Rotation” on page 2-59](#)
- [“Loop Vectorization” on page 2-62](#)
- [“Modulo Scheduling” on page 2-64](#)

The following examples are presented in terms of a hypothetical machine. This machine is capable of issuing up to two instructions in parallel, provided one instruction is an arithmetic instruction, and the other is a load or a store. Two arithmetic instructions may not be issued at once, nor may two memory accesses:

```
t0 = t0 + t1;           // valid: single arithmetic
t2 = [p0];              // valid: single memory access
[p1] = t2;              // valid: single memory access
t2 = t1 + 4, t1 = [p0];  // valid: arithmetic and memory
t5 += 1, t6 -= 1;        // invalid: two arithmetic
[p3] = t2, t4 = [p5];    // invalid: two memory
```

Achieving Optimal Performance from C/C++ Source Code

The machine can use the old value of a register and assign a new value to it in the same cycle, for example:

```
t2 = t3 + 4, t2 = [p2];    // valid: arithmetic and memory
```

The value of `t1` on entry to the instruction is the value used in the addition. On completion of the instruction, `t1` contains the value loaded via the `p0` register.

The examples will show “START LOOP N” and “END LOOP”, to indicate the boundaries of a loop that iterates N times. (The mechanisms of the loop entry and exit are not relevant).

Software Pipelining

“Software pipelining” is analogous to hardware pipelining used in some processors. Whereas hardware pipelining allows a processor to start processing one instruction before the preceding instruction has completed, software pipelining allows the generated code to begin processing the next iteration of the original source-code loop before the preceding iteration is complete.

Software pipelining makes use of a processor's ability to multi-issue instructions. Regarding known delays between instructions, it schedules instructions from later iterations where there is spare capacity.

Loop Rotation

“Loop rotation” is a common technique of achieving software pipelining. It changes the logical start and end positions of the loop within the overall instruction sequence, to allow a better schedule within the loop itself. For example, this loop:

```
START LOOP N  
A  
B  
C
```

How Loop Optimization Works

```
D
E
END LOOP
```

could be rotated to produce the following loop:

```
A
B
C
START LOOP N-1
D
E
A
B
C
END LOOP
D
E
```

The order of instructions in the loop kernel is now different. It still circles from instruction E back to instruction A, but now it starts at D, rather than A. The loop also has a prolog and epilog added, to preserve the intended order of instructions. Since the combined prolog and epilog make up a complete iteration of the loop, the kernel is now executing $N-1$ iterations, instead of N .

In this example, consider the following loop:

```
START LOOP N
t0 += 1
[p0++] = t0
END LOOP
```

This loop has a two-cycle kernel. While the machine could execute the two instructions in a single cycle – an arithmetic instruction and a memory access instruction – to do so would be invalid, because the second instruction depends upon the value computed in the first instruction.

Achieving Optimal Performance from C/C++ Source Code

However, if the loop is rotated, we get:

```
t0 += 1
START LOOP N-1
[p0++] = t0
t0 += 1
END LOOP
[p0++] = t0
```

The value being stored is computed in the previous iteration (or before the loop starts, in the prolog). This allows the two instructions to be executed in a single cycle:

```
t0 += 1
START LOOP N-1
[p0++] = t0, t0 += 1
END LOOP
[p0++] = t0
```

Rotating the loop has presented an opportunity by which the k th iteration of the original loop is starting ($t0 += 1$) while the $(k-1)$ th iteration is completing ($[p0++] = t0$), so rotation has achieved software pipelining, and the performance of the loop is doubled.

Notice that this process has changed the structure of the program slightly: suppose that the loop construct always executes the loop at least once; that is, it is a $1..N$ count. Then if $N==1$, changing the loop to be $N-1$ would be problematic. In this example, the compiler inserts a guard: a conditional jump around the loop construct for the circumstances where the compiler cannot guarantee that $N > 1$:

```
t0 += 1
IF N == 1 JUMP L1;
START LOOP N-1
[p0++] = t0, t0 += 1
END LOOP
L1:
[p0++] = t0
```

Loop Vectorization

“Loop vectorization” is another transformation that allows the generated code to execute more than one iteration in parallel. However, vectorization is different from software pipelining. Where software pipelining uses a different ordering of instructions to get better performance, vectorization uses a different set of instructions. These vector instructions act on multiple data elements concurrently to replace multiple executions of each original instruction.

For example, consider this dot-product loop:

```
int i, sum = 0;
for (i = 0; i < n; i++) {
    sum += x[i] * y[i];
}
```

This loop walks two arrays, reading consecutive values from each, multiplying them and adding the result to the on-going sum. This loop has these important characteristics:

- Successive iterations of the loop read from adjacent locations in the arrays.
- The dependency between successive iterations is the summation, a commutative operation.
- Operations such as load, multiply and add are often available in parallel versions on embedded processors.

These characteristics allow the optimizer to vectorize the loop so that two elements are read from each array per load, two multiplies are done, and two totals maintained.

The vectorized loop would be:

```
t0 = t1 = 0
START LOOP N/2
```

Achieving Optimal Performance from C/C++ Source Code

```
t2 = [p0++] (Wide)    // load x[i] and x[i+1]
t3 = [p1++] (Wide)    // load y[i] and y[i+1]
t0 += t2 * t3 (Low), t1 += t2 * t3 (High)    // vector mulacc
END LOOP
t0 = t0 + t1           // combine totals for low and high
```

Vectorization is most efficient when all the operations in the loop can be expressed in terms of parallel operations. Loops with conditional constructs in them are rarely vectorizable, because the compiler cannot guarantee that the condition will evaluate in the same way for all the iterations being executed in parallel.

Vectorization is also affected by data alignment constraints and data access patterns. Data alignment affects vectorization because processors often constrain loads and stores to be aligned on certain boundaries. While the unvectorized version will guarantee this, the vectorized version imposes a greater constraint that may not be guaranteed. Data access patterns affect vectorization because memory accesses must be contiguous. If a loop accessed every tenth element, for example, then the compiler would not be able to combine the two loads for successive iterations into a single access.

Vectorization divides the generated iteration count by the number of iterations being processed in parallel. If the trip count of the original loop is unknown, the compiler will have to conditionally execute some iterations of the loop.



Vectorization and software pipelining are not mutually exclusive: the compiler may vectorize a loop and then use software pipelining to obtain better performance.

Modulo Scheduling

Loop rotation, as described earlier, is a simple software-pipelining method that can often improve loop performance, but more complex examples require a more advanced approach. The compiler uses a popular technique known as “Modulo Scheduling” which can produce more efficient schedules for loops than simple loop rotation.

Modulo scheduling is used to schedule innermost loops without control flow. A modulo-scheduled loop is described using the following parameters:

- Initiation interval (*II*): the number of cycles between initiating two successive iterations of the original loop.
- Minimum initiation interval due to resources (*res MII*): a lower limit for the initiation interval (*II*); an *II* lower than this would mean at least one of the resources being used at greater capacity than the machine allows.
- Minimum initiation interval due to recurrences (*rec MII*): an instruction cannot be executed until earlier instruction on which it depends have also been executed. These earlier instructions may belong to a previous loop iteration. A cycle of such dependencies (a *recurrence*) imposes a minimum number of cycles for the loop.
- Stage count (*SC*): the number of initiation intervals until the first iteration of the loop has completed. This is also the number of iterations in progress at any time within the kernel.
- Modulo variable expansion unroll factor (*MVE unroll*): the number of times the loop has to be unrolled to generate the schedule without overlapping register lifetimes.
- Trip count: the number of times the loop kernel iterates.
- Trip modulo: a number that is known to divide the trip count.

Achieving Optimal Performance from C/C++ Source Code

- Trip maximum: an upper limit for the trip count.
- Trip minimum: a lower limit for the trip count.

Understanding these parameters will allow you to interpret the generated code more easily. The compiler's assembly annotations use these terms, so you can examine the source code and the generated instructions, to see how the scheduling relates to the original source. See [“Assembly Optimizer Annotations” on page 2-80](#) for more information.

Modulo scheduling performs software pipelining by:

- Ordering the original instructions in a sequence (for simplicity referred to as the “*base schedule*”) that can be repeated after an interval known as the “*initiation interval*” (“II”);
- Issuing parts of the base schedule belonging to successive iterations of the original loop, in parallel.

For the purposes of this discussion, all instructions will be assumed to require only a single cycle to execute; on a real processor, stalls affect the initiation interval, so a loop that executes in II cycles may have fewer than II instructions.

Initiation Interval (II) and the kernel

Consider the loop

```
START LOOP N
A
B
C
D
E
F
G
H
END LOOP
```

How Loop Optimization Works

Now consider that the compiler finds a new order for A, B, C, D, E, F, G, H grouping; some of them on the same cycle so that a new instance of the sequence can be started every two cycles. Say this *base schedule* is given in [Table 2-3](#) where I1, I2, ..., I8 are A, B, ..., H reordered. Albeit a valid schedule for the original loop, the base schedule is not the final modulo schedule; it may not even be the shortest schedule of the original loop. However, the base schedule is used to obtain the modulo schedule, by being able to initiate it every $II=2$ cycles, as seen in [Table 2-4](#).

Table 2-3. Base Schedule

Cycle	Instructions
1	I1
2	I2, I3
3	I4, I5
4	I6
5	I7
6	I8

Table 2-4. Obtaining the Modulo Schedule by Repeating the Base Schedule every $II=2$ Cycles

Cycle	Iteration 1	Iteration 2	Iteration 3	Iteration 4
1	I1			
2	I2, I3			
3	I4, I5	I1		
4	I6	I2, I3		
5	I7	I4, I5	I1	
6	I8	I6	I2, I3	
7		I7	I4, I5	I1
8		I8	I6	I2, I3

Table 2-4. Obtaining the Modulo Schedule by Repeating the Base Schedule every $II=2$ Cycles (Cont'd)

Cycle	Iteration 1	Iteration 2	Iteration 3	Iteration 4
9			I7	I4, I5
10			I8	I6

Starting at cycle 5, the pattern in [Table 2-5](#) keeps repeating every 2 cycles. This repeating pattern is the *kernel*, and it represents the modulo scheduled loop.

Table 2-5. Loop Kernel, $N \geq 3$

Cycle	Iteration N-2 (last stage)	Iteration N-1 (2nd stage)	Iteration N (1st stage)
$II*N-1$	I7	I4, I5	I1
$II*N$	I8	I6	I2, I3

The initiation interval has the value $II=2$, because iteration $i+1$ can start two cycles after the cycle on which iteration i starts. This way, one iteration of the original loop is initiated every II cycles, running in parallel with previous, unfinished iterations.

The initiation interval of the loop indicates several important characteristics of the schedule for the loop:

- The loop kernel will be II cycles in length.
- A new iteration of the original loop will start every II cycles. An iteration of the original loop will end every II cycles.
- The same instruction will execute on cycle c and on cycle $c+II$ (hence the name *modulo schedule*).

Finding a modulo schedule implies finding a base schedule and an II such that the base schedule can be initiated every II cycles.

How Loop Optimization Works

If the compiler can reduce the value for II, it can start the next iteration sooner, and thus increase the performance of the loop: The lower the II, the more efficient the schedule. However the II is limited by a number of factors, including:

- The machine resources required by the instructions in the loop.
- The data dependencies and stalls between instructions.

We'll examine each of these limiting factors.

Minimum Initiation Interval Due to Resources (Res MII)

The first factor that limits II is machine resource usage. Let's start with the simple observation that the kernel of a modulo scheduled loop contains the same set of instructions as the original loop.

Assume a machine that can execute up to four instructions in parallel. If the loop has 8 instructions, then it requires a minimum of 2 lines in the kernel, since there can be at most 4 instructions on a line. This implies II has to be at least 2, and we can tell this without having found a base schedule for the loop, or even knowing what the specific instructions are.

Consider another example where the original loop contains 3 memory accesses to be scheduled on a machine that supports at most 2 memory accesses per cycle. This implies at least 2 cycles in the kernel, regardless of the rest of the instructions.

Given a set of instructions in a loop, we can determine a lower bound for the II of any modulo schedule for that loop based on resources required. This lower bound is called the “*Resource based Minimum Initiation Interval*” (*Res MII*)

Minimum Initiation Interval Due to Recurrences (Rec MII)

A less obvious limitation for finding a low II are cycles in the data dependencies between instructions.

Assume that the loop to be scheduled contains (among others) the instructions:

```
i3: t3=t1+t5;    // t5 carried from the previous iteration
i5: t5=t1+t3;
```

Assume each line of instructions takes 1 cycle. If *i3* is executed at cycle *c* then *t3* is available at cycle *c*+1 and *t5* cannot be computed earlier than *c*+1 (because it depends on *t3*), and similarly the next time we compute *t3* cannot be earlier than *c*+2. Thus if we execute *i3* at cycle *c*, the next time we can execute *i3* again cannot be earlier than *c*+2. But for any modulo schedule, if an instruction is executed at cycle *c*, the next iteration will execute the same instruction at cycle *c*+*II*. Therefore, *II* has to be at least 2 due to the circular data dependency path *t3*->*t5*->*t3*.

This lower bound for *II*, given by circular data dependencies (recurrences) is called the “*Minimum Initiation Interval Due to Recurrences*” (*Rec MII*), and the data dependency path is called “*loop carry path*”. There can be any number of loop carry paths in a loop, including none, and they are not necessarily disjoint.

Stage Count (SC)

The kernel in [Table 2-5](#) is formed of instructions which belong to 3 distinct iterations of the original loop: {*I7*, *I8*} end the “oldest” iteration — in other words they belong to the iteration started the longest time before the current cycle; {*I4*, *I5*, *I6*} belong to the next oldest initiated iteration, and so on. {*I1*, *I2*, *I3*} are the beginning of the youngest iteration.

The number of iterations of the original loop in progress at any time within the kernel is called the “*Stage Count*” (*SC*). This is also the number of initiation intervals until the first iteration of the loop completes. In our example *SC*=3.

How Loop Optimization Works

The final schedule requires peeling a few instructions (the prolog) from the beginning of the first iteration and a few instructions (the epilog) from the end of the last iteration in order to preserve the structure of the kernel. This reduces the trip count from N to $N-(SC-1)$:

```
I1;                                // prolog
I2,I3;                             // prolog
I4,I5,      I1;                    // prolog
I6,          I2,I3;                // prolog
LOOP N-2                            // i.e. N-(SC-1), where SC=3
I7,          I4,I5,      I1;        // kernel
I8,          I6,          I2,I3;    // kernel
END LOOP

          I7,          I4, I5;      // epilog
          I8,          I6;          // epilog
          I7;                  // epilog
          I8;                  // epilog
```

Another way of viewing the modulo schedule is to group instructions into stages as in [Figure 2-2](#), where each stage is viewed as a vector of height $II=2$ of instruction lists (that represent parts of instruction lines).

Figure 2-2. Instructions Grouped into Stages

StageCount	Instructions
SC0	I1, I2, I3
SC1	I4, I5, I6
SC2	I7, I8

Achieving Optimal Performance from C/C++ Source Code

Now the schedule can be viewed as:

```
SC0                                // prolog
SC1    SC0                        // prolog
LOOP (N-2)                        // That is N-(SC-1), where SC=3
SC2    SC1    SC0                // kernel
        SC2    SC1    SC0        // kernel
END LOOP
        SC2    SC1                // epilog
        SC2                        // epilog
```

where, for example, SC2 SC1 is the 2 line vector obtained from concatenating the lists in SC2 and SC1.

Variable Expansion and MVE Unroll

There is one more issue to address for modulo schedule correctness.

Table 2-6. Problematic Instance

Generic instruction	Specific instance
I1	t1=[p1++]
I2	t2=[p2++]
I3	t3=t1+t5
I4	t4=t2+1
I5	t5=t1+t3
I6	t6=t4*t5
I7	t7=t6*t3
I8	[p8++]=t7

Consider the sequence of instructions in [Table 2-6](#). [Table 2-7](#) shows the base schedule that is an instance of the one in [Table 2-3](#), and [Table 2-8](#) shows the corresponding modulo schedule with $II=2$.

How Loop Optimization Works

Table 2-7. Base Schedule from [Table 2-3](#) applied to the Instances in [Table 2-6](#)

1	$t1 = [p1++]$
2	$t2 = [p2++]$, $t3 = t1 + t5$
3	$t4 = t2 + 1$, $t5 = t1 + t3$
4	$t6 = t4 * t5$
5	$t7 = t6 * t3$
6	$[p8++] = t7$

Table 2-8. Modulo Schedule Broken by Overlapping Lifetimes of $t3$

	Iteration 1	Iteration 2	Iteration 3 ...
1	$t1 = [p1++]$		
2	$t2 = [p2++]$, $t3 = t1 + t5$		
3	$t4 = t2 + 1$, $t5 = t1 + t3$	$t1 = [p1++]$	
4	$t6 = t4 * t5$	$t2 = [p2++]$, $t3 = t1 + t5$	
5	$t7 = t6 * t3$	$t4 = t2 + 1$, $t5 = t1 + t3$	$t1 = [p1++]$
6	$[p8++] = t7$	$t6 = t4 * t5$	$t2 = [p2++]$, $t3 = t1 + t5$
7		$t7 = t6 * t3$	$t4 = t2 + 1$, $t5 = t1 + t3$
8		$[p8++] = t7$	$t6 = t4 * t5$
9			$t7 = t6 * t3$
10			$[p8++] = t7$

However, there is a problem with the schedule in [Table 2-8](#): $t3$ defined in the fourth cycle (second column in the table) is used on the fifth cycle (first column); however, the intended use was of the value defined on the second cycle (first column). In general, the value of $t3$ used by $t7 = t6 * t3$ in the kernel will be the one defined in the previous cycle, instead of the one defined 3 cycles earlier, as intended. Thus, if the compiler were to use this

schedule as-is, it would be clobbering the live value in $t3$. The lifetime of each value loaded into $t3$ is 3 cycles, but the loop's initiation interval is only 2, so the lifetimes of $t3$ from different iterations overlap.

The compiler fixes this by duplicating the kernel as many times as needed to exceed the longest lifetime in the base schedule, then renaming the variables that clash – in this case, just $t3$. In [Table 2-9](#), we see that the length of the new loop body is 4, greater than the lifetimes of the values in the loop.

Table 2-9. Modulo Schedule Corrected by Variable Expansion: $t3$ and $t3_2$

	Iteration 1	Iteration 2	Iteration 3	Iteration 4 ...
1	$t1=[p1++]$			
2	$t2=[p2++]$, $t3=t1+t5$			
3	$t4=t2+1$, $t5=t1+t3$	$t1=[p1++]$		
4	$t6=t4*t5$	$t2=[p2++]$, $t3_2=t1+t5$		
5	$t7=t6*t3$	$t4=t2+1$, $t5=t1+t3_2$	$t1=[p1++]$	
6	$[p8++]=t7$	$t6=t4*t5$	$t2=[p2++]$, $t3=t1+t5$	
7		$t7=t6*t3_2$	$t4=t2+1$, $t5=t1+t3$	$t1=[p1++]$
8		$[p8++]=t7$	$t6=t4*t5$	$t2=[p2++]$, $t3_2=t1+t5$
9			$t7=t6*t3$	$t4=t2+1$, $t5=t1+t3_2$
10			$[p8++]=t7$	$t6=t4*t5$
11				$t7=t6*t3_2$
12				$[p8++]=t7$

How Loop Optimization Works

So the loop becomes:

```
t1=[p1++];
t2=[p2++],t3=t1+t5;
t4=t2+1,t5=t1+t3,  t1=[p1++];
t6=t4*t5,          t2=[p2++],t3_2=t1+t5;
LOOP (N-2)/2
t7=t6*t3,          t4=t2+1,t5=t1+t3_2,  t1=[p1++];
[p8++]=t7,         t6=t4*t5,          t2=[p2++],t3=t1+t5;
                t7=t6*t3_2,          t4=t2+1,t5=t1+t3,  t1=[p1++];
                [p8++]=t7,          t6=t4*t5, t2=[p2++],t3_2=t1+t5;

END LOOP
                t7=t6*t3,          t4=t2+1,t5=t1+t3_2;
                [p8++]=t7,          t6=t4*t5;
                                t7=t6*t3_2;
                                [p8++]=t7;
```

This process of duplicating the kernel and renaming colliding variables is called *variable expansion*, and the number of times the compiler duplicates the kernel is referred to as the *modulo variable expansion factor (MVE)*.

Conceptually we use different set of names, “register sets”, for successive iterations of the original loop in progress in the unrolled kernel (in practice we rename just the conflicting variables, see [Table 2-10](#)). In terms of reading the code, this means that a single iteration of the loop generated by the compiler will be processing more than one iteration of the original loop. Also, the compiler will be using more registers to allow the iterations of the original loop to overlap without clobbering the live values.

In terms of stages:

```
SC0                                // prolog
SC1      SC0_2                    // prolog
LOOP (N-2)/2                      // That is N-(SC-1)/MVE, where SC=3, MVE=2
SC2      SC1_2      SC0           // kernel
          SC2_2      SC1      SC0_2 // kernel
END LOOP
          SC2      SC1_2 // epilog
          SC2_2      // epilog
```

Achieving Optimal Performance from C/C++ Source Code

where SCN_2 is SCN subject to renaming; in our case only occurrences of $t3$ are renamed as $t3_2$ in SCN_2 .

In terms of instructions:

```
I1;                                // prolog
I2,I3;                             // prolog
I4,I5,      I1_2;                   // prolog
I6,          I2_2,I3_2;              // prolog
LOOP(N-2)/2      // That is N-(SC-1)/MVE, where SC=3, MVE=2
I7,          I4_2,I5_2,  I1;         // kernel
I8,          I6_2,      I2,I3;       // kernel
          I7_2,      I4,I5,      I1_2; // kernel
          I8_2,      I6,          I2_2,I3_2; // kernel
END LOOP
          I7,          I4_2,I5_2; // epilog
          I8,          I6_2;      // epilog
          I7_2;           // epilog
          I8_2;           // epilog
```

where IN_2 is IN subject to renaming, in our case only occurrences of $t3$ are renamed as $t3_2$ in all IN_2 , as seen in [Table 2-10](#).

Table 2-10. Instructions after Modulo Variable Expansion

Generic instruction	Specific instance
I1 and I1_2	$t1=[p1++]$
I2 and I2_2	$t2=[p2++]$
I3	$t3=t1+t5$
I3_2	$t3_2=t1+t5$
I4 and I4_2	$t4=t2+1$
I5	$t5=t1+t3$
I5_2	$t5=t1+t3_2$
I6 and I6_2	$t6=t4*t5$

How Loop Optimization Works

Table 2-10. Instructions after Modulo Variable Expansion (Cont'd)

Generic instruction	Specific instance
I7 and I7_2	$t7 = t6 * t3$
I8 and I8_2	$[p8++] = t7$

Trip Count

Notice that as the modulo scheduler expands the loop kernel to add in the extra variable sets, the iteration count of the generated loop changes from $(N-SC)$ to $(N-SC)/MVE$. This is because each iteration of the generated loop is now doing more than one iteration of the original loop, so fewer generated iterations are required.

However, this also relies on the compiler knowing that it can divide the loop count in this manner. For example, if the compiler produces a loop with $MVE=2$ so that the count should be $(N-SC)/2$, an odd value of $(N-SC)$ causes problems. In these cases, the compiler generates additional “peeled” iterations of the original loop to handle the remaining iteration. As with rotation, if the compiler cannot determine the value of N , it will make parts of the loop—the kernel or peeled iterations—conditional so that they are executed only for the appropriate values of N .

The number of times the generated loop iterates is called the “trip count”. As explained above, sometimes knowing the trip count is important for efficient scheduling. However, the trip count is not always available. Lacking it, additional information may be inferred, or passed to the compiler through the `loop_count` pragma, specifying:

- “*Trip modulo*”: a number known to divide the trip count
- “*Trip minimum*”: a lower bound for the trip count
- “*Trip maximum*”: an upper bound for the trip count

A Worked Example

The following floating-point scalar product loop are used to show how the compiler optimizer works.

Example: C source code for floating-point scalar product.

```
float sp(float *a, float *b, int n) {
    int i;
    float sum=0;
    __builtin_aligned(a, 2);
    __builtin_aligned(b, 2);
    for (i=0; i<n; i++) {
        sum+=a[i]*b[i];
    }
    return sum;
}
```

After code generation and conventional scalar optimizations, the compiler generates a loop that resembles the following example.

Example: Initial code generated for floating-point scalar product

```
lcntr = r3, do(pc, .P1L10-1)until lce;
.P1L9:
r4 = dm(i1, m6);
r2 = dm(i0, m6);
f12 = f2 * f4;
f10 = f10 + f12;
// end_loop .P1L9;
.P1L10:
```

The loop exit test has been moved to the bottom and the loop counter rewritten to count down to zero. This enables a zero-overhead hardware loop to be created. (r3 is initialized with the loop count.) `sum` is being accumulated in `r10`. `i0` and `i1` hold pointers that are initialized with the parameters `a` and `b` and incremented on each iteration.

How Loop Optimization Works

The ADSP-2116x, ADSP-2126x and ADSP-2136x processors have two compute units that may perform computations simultaneously. To use both these compute blocks, the optimizer unrolls the loop to run two iterations in parallel. `sum` is now being accumulated in `r10` and `s10`, which must be added together after the loop to produce the final result. To use the dual-word loads needed for the loop to be as efficient as this, the compiler has to know that `i0` and `i1` have initial values that are even. This is done in the above example by use of `__builtin_aligned`, although it could also be propagated with IPA.

Note also that unless the compiler knows that original loop was executed an even number of times, a conditionally-executed odd iteration must be inserted outside the loop. `r3` is now initialized with half the value of the original loop.

Example: Code generated for floating-point scalar product after vectorization transformation

```
bit set model 0x200000; nop;          // enter SIMD mode
m4 = 2;
lcntr = r3, do(pc, .P1L10-1)until lce;
.P1L9:
r4 = dm(i1, m4);
r2 = dm(i0, m4);
f12 = f2 * f4;
f10 = f10 + f12;
// end_loop .P1L9;
.P1L10:
bit clr model 0x200000; nop;          // exit SIMD mode
```

Finally, the optimizer rotates the loop, unrolling and overlapping iterations to obtain highest possible use of functional units. Code similar to the following is generated, if it were known that the loop was executed at least four times and the loop count was a multiple of two.

Example: Code generated for floating-point scalar product after software pipelining

```
    bit set model 0x200000; nop;          // enter SIMD mode
    m4 = 2;
    r4 = dm(i1, m4);
    r2 = dm(i0, m4);
    lcntr = r3, do(pc, .P1L10-1)until lce;
.P1L9:
    f12 = f2 * f4, r4 = dm(i1, m4);
    f10 = f10 + f12, r2 = dm(i0, m4);
    // end_loop .P1L9;
.P1L10:
    f12 = f2 * f4;
    f10 = f10 + f12;
    bit clr model 0x200000; nop;          // exit SIMD mode
```

If the original source code is amended to declare one of the pointers with the pm qualifier, the following optimal code is produced for the loop kernel.

Example: Code generated for floating-point scalar product when one buffer placed in PM

```
    bit set model 0x200000; nop;          // enter SIMD mode
    m4 = 2;
    r5 = pm(i1, m4);
    r2 = dm(i0, m4);
    r4 = pm(i1, m4);
    f12 = f2 * f5, r2 = dm(i0, m4);
    lcntr = r3, do(pc, .P1L10-1)until lce;
.P1L9:
    f12 = f2 * f4, f10 = f10 + f12, r2 = dm(i0, m4), r4 = pm(i1, m4);
    // end_loop .P1L9;
.P1L10:
    f12 = f2 * f4, f10 = f10 + f12;
    f10 = f10 + f12;
    bit clr model 0x200000; nop;          // exit SIMD mode
```

Assembly Optimizer Annotations

When the compiler optimizations are enabled, the compiler can perform a large number of optimizations to generate the resultant assembly code. The decisions taken by the compiler as to whether certain optimizations are safe or worthwhile are generally invisible to a programmer. However, it could be beneficial to get feedback from the compiler regarding the decisions made during optimization. The intention of the information provided is to give a programmer an understanding of how close to optimal a program is and what more could possibly be done to improve the generated code.

The feedback from the compiler optimizer is provided by means of annotations made to the assembly file generated by the compiler. The assembly file generated by the compiler can be kept by specifying the `-S` switch (on page 1-61), the `-save-temps` switch (on page 1-61) or by checking the **Project Options->Compile->General->Save temporary files** option in VisualDSP++ IDDE.

The assembly code generated by the compiler optimizer is annotated with the following information:

- “Global Information” on page 2-81
- “Procedure Statistics” on page 2-82
- “Instruction Annotations” on page 2-87
- “Loop Identification” on page 2-87
- “Vectorization” on page 2-94
- “Modulo Scheduling Information” on page 2-98
- “Warnings, Failure Messages and Advice” on page 2-105

The assembly annotations provide information in several areas that you can use to assist the compiler’s evaluation of your source code. In turn, this improves the generated code. For example, annotations could provide

indications of resource usage or the absence of a particular optimization from the resultant code. Annotations which note the absence of optimization can often be more important than those noting its presence. Assembly code annotations give the programmer insight into why the compiler enables and disables certain optimizations for a specific code sequence.

The assembly output for the examples in this chapter may differ based on optimization flags and the version of the compiler. As a result, you may not be able to reproduce these results exactly.

Global Information

For each compilation unit, the assembly output is annotated with:

- The time of the compilation
- The options used during that compilation.
- The architecture for which the file was compiled.
- The silicon revision used during the compilation
- A summary of the workarounds associated with the specified architecture and silicon revision. These workarounds are divided into:
 - Disabled: the workarounds that were not applied
 - Enabled: the workarounds that were applied during the compilation.
 - Always on: the workarounds that are always applied and that cannot be disabled, not even by using the `-si-revision none` compiler switch.

For instance, if the file `hello.c` is compiled at 11am, on June 28 using the following command line:

```
cc21k -O -S hello.c
```

then the `hello.s` file will show:

```
.file "hello.c";  
//Compilation time: Thu Jun 28 11:00:00 2007  
//Compiler options: -O -S  
//Architecture: ADSP-21060  
//Silicon revision: 3.1  
//Anomalies summary:  
// Disabled:w_anomaly_45,w_rframe,w_swfa, ...  
// Always on: w_end_of_loop
```

Procedure Statistics

For each function, the following is reported:

- Frame size: size of stack frame.
- Registers used. Since function calls tend to implicitly clobber registers, there are several sets:
 - The first set is composed of the scratch registers changed by the current function. This does not count the registers that are implicitly clobbered by the functions called from the current function.
 - The second set are the call-preserved registers changed by the current function. This does not count the registers that are implicitly clobbered by the functions called from the current function.
 - The third set are the registers clobbered by the inner function calls.
- Inlined Functions – if inlining happens, then the header of the caller function reports which functions were inlined inside it and where. Each inlined function is reported using the position of the

inlined call. All the functions inlined inside the inlined function are reported as well, generating a tree of inlined calls. Each node, except the root, has the form:

file_name:line:column'function_name

where:

- *function_name* is the name of the function inlined.
- *line* is the line number of the call to *function_name*, in the source file.
- *column* is the column number of the call to *function_name*, in the source file.
- *file_name* is the name of the source file calling *function_name*.

Example A (Procedure Statistics)

Consider the following program:

```
struct str {
    int x1, x2;
};
int func1(struct str*, int *);
int func2(struct str s);
int foo(int in)
{
    int sum = 0;
    int local;
    struct str l_str;
    sum += func1(&l_str, &local);
    sum += func2(l_str);
    return sum;
}
```

Assembly Optimizer Annotations

The procedure statistics for `foo` are:

```
_foo:
.LN_foo:
//-----
// Procedure statistics:
// Frame size           = 7 words
// Scratch registers used:{r0,r2,r4,r8,i12,acc}
// Call preserved registers used:{r15,i5,i7}
// Registers that could be clobbered by function
calls:{r0-r2,r4,r8,r12,i4,i12-i13,b4,b12-b13,m4,m12,acc,mcc,scc,
btf,sky,lcntr,px1-px2}
//-----
// line "ExampleA.c":7
    r2=i5;
// line 11
    i5=i6;
    // -- stall --
    modify(i5,-4);
    r4=i5;
// line 7
    modify(i7,-5);
// line 11
    modify(i5,2);
// line 7
    dm(-6,i6)=r15;
    dm(-5,i6)=r2;
// line 11
    r8=i5;
    cjump _func1 (db); dm(i7,m7)=r2; dm(i7,m7)=pc;
// line 12
    r15=pass r0, modify(i5,m7);
    r2=dm(i5,m5);
    dm(i7,m7)=r2;
    r2=dm(m7,i5);
    dm(i7,m7)=r2;
    cjump _func2 (db); dm(i7,m7)=r2; dm(i7,m7)=pc;
```


Achieving Optimal Performance from C/C++ Source Code

```
        modify(i7,2);
        r0=r15+r0, i12=dm(m7,i6);
// line 13
        i5=dm(-5,i6);
        // -- stall --
        r15=dm(-6,i6);
        jump (m14,i12) (db); rframe; nop;
.LN._foo.end:
```

Notes:

The following notes apply to procedure statistics:

- The frame size is 7 words, indicating how much space is allocated on the stack by the function. The frame size includes:
 - one word for the old frame pointer
 - one word for the return address
 - the space allocated by the compiler, for local variables (3 words: one for `local`, 2 for `l_str`)
 - space required to save any callee-preserved registers (one word for each of `R15`, `i5`)
 - space required for parameters being passed to functions called by this one (none in this case)
- The set of scratch registers modified is `{r0,r2,r4,r8,i12,acc}` because, except for the `func1` and `func2` function calls, these are the only scratch registers changed by `foo`.
- The set of call preserved registers used is `{r15,i5,i7}` because these are the only call preserved registers used by `foo`.
- The set of registers clobbered by function calls contains the set of registers potentially changed by the calls to `func1` and `func2`.

Example B (Inlining Summary)

This is an example of inlined function reporting.

```
1 void f4(int n);
2 __inline void f3(int n)
3 {
4     f4(n);
5 }
6
7 __inline void f2(int n)
8 {
9     while (n--) {
10         f3(n);
11         f3(2*n);
12     }
13 }
14 void f1(volatile unsigned int i)
15 (
16     f2(30);
17 )
```

f1 inlines the call of f2, which inlines the call of f3 in two places. The procedure statistics for f1 reports these inlined calls:

```
_f1:
//-----
// Procedure statistics
. . . . .
// Inlined in _f1:
//     ExampleB.c:16:7'_f2
//         ExampleB.c:11:11'_f3
//         ExampleB.c:10:11'_f3
//-----
. . . . .
```

f1 reports that f2 was inlined at line 16 (column 7) and, implicitly, f1 also inlined the two calls of f3 inside f2.

Instruction Annotations

Sometimes the compiler annotates certain assembly instructions. It does so in order to point to possible inefficiencies in the original source code, or when the `-annotate-loop-instr` compiler switch ([on page 1-26](#)) is used to annotate the instructions related to modulo scheduled loops.

The format of an assembly line containing several instructions is changed. Instructions issued in parallel are no longer shown all on the same assembly line; each is shown on a separate assembly line, so that the instruction annotations can be placed after the corresponding instructions. Thus

```
instruction_1, instruction_2, instruction_3;
```

is displayed as:

```
instruction_1, // {annotations for instruction_1}
instruction_2, // {annotations for instruction_2}
instruction_3; // {annotations for instruction_3}
```

Loop Identification

One useful annotation is loop identification—that is, showing the relationship between the source program loops and the generated assembly code. This is not easy due to the various loop optimizations. Some of the original loops may not be present, because they are unrolled. Other loops get merged, making it difficult to describe what has happened to them.

The assembly code generated by the compiler optimizer is annotated with the following loop information:

- [“Loop Identification Annotations” on page 2-88](#)
- [“File Position” on page 2-91](#)

Finally, the assembly code may contain compiler-generated loops that do not correspond to any loop in the user program, but rather represent constructs such as structure assignment or calls to `memcpy`.

Loop Identification Annotations

Loop identification annotation rules are:

- Annotate only the loops that originate from the C looping constructs `do`, `while`, and `for`. Therefore, any `goto` defined loop is not accounted for.
- A loop is identified by the position of the corresponding keyword (`do`, `while`, `for`) in the source file.
- Account for all such loops in the original user program.
- Generally, loop bodies are delimited between the `Lx: Loop at <file position>` and `End Loop Lx` assembly annotation. The former annotation follows the label of the first block in the loop. The later annotation follows the jump back to the beginning of the loop. However, there are cases in which the code corresponding to a user loop cannot be entirely represented between such two markers. In such cases the assembly code contains blocks that belong to a loop, but are not contained between that loop's end markers. Such blocks are annotated with a comment identifying the innermost loop they belong to, `Part of Loop Lx`.
- Sometimes a loop in the original program does not show up in the assembly file, because it was either transformed or deleted. In either case, a short description of what happened to the loop is given at the beginning of the function.
- A program's innermost loops are those loops that do not contain other loops. In addition to regular loop information, the innermost loops with no control flow and no function calls are annotated with additional information such as:
 - **Cycle count.** The number of cycles needed to execute one iteration of the loop, including the stalls.

- **Resource usage.** The resources used during one iteration of the loop. For each resource we show how many of that resource are used, how many are available and the percentage of utilization during the entire loop. Resources are shown in decreasing order of utilization. Note that 100% utilization means that the corresponding resource is used at its full capacity and represents a bottleneck for the loop.
- **Register usage.** If the `-annotate-loop-instr` compilation flag (switch) is used, then the register usage table is shown. This table has one column for every register that is defined or used inside the loop. The header of the table shows the names of the registers, written on the vertical, top down. The registers that are not accessed do not show up. The columns are grouped on data registers, pointer registers and all other registers. For every cycle in a loop (including stalls), there is a row in the array. The entry for a register has a '*' on that row if the register is either live or being defined at that cycle.

If the code executes in parallel (in a SIMD region), accessing a D register usually means accessing its corresponding shadow register in parallel. In these cases, the name of the register is prefixed with 2x. For instance, `2xr2`.

- **Optimizations.** Some loops are subject to optimizations such as vectorization or modulo scheduling. These loops receive additional annotations as described in the vectorization and modulo scheduling paragraphs.
- Sometimes the compiler generates additional loops that may or may not be directly associated with the loops in the user program. Whenever possible, the compiler annotations try to show the relation between such compiler-generated loops and the original source code.

Example C (Loop Identification, for ADSP-21060 Processor)

Consider the following example:

```
1 int bar(int a[10000])
2 {
3     int i, sum = 0;
4     for (i = 0; i < 9999; ++i)
5         sum += (sum + 1);
6     while (i-- < 9999) /* this loop doesn't get executed */
7         a[i] = 2*i;
8     return sum;
9 }
```

The two loops are accounted for as follows:

```
_bar:
.LN_bar:
//-----
..... procedure statistics .....

//-----
// Original Loop at "ExampleC.c" line 6 col 3 -- loop structure
// removed due to constant propagation.
//-----
        r0=m5;
// line "ExampleC.c":4
        lcntr=9999, do (pc,.P34L13-1) until lce;
.P34L2:
//-----
//   Loop at "ExampleC.c" line 4 col 3
//-----
..... loop annotations .....
//-----
// line 5
        r2=r0+1;
        r0=r2+r0;
// line 4
        // end loop .P34L2;
//-----
//   End Loop L2
```

```
//-----  
.P34L13:  
//-----  
//   Part of top level (no loop)  
//-----  
// line 8  
        i12=dm(m7,i6);  
        jump (m14,i12) (db); rframe; nop;  
.LN._bar.end:
```

Notes:

The following notes apply to loop identification annotations:

- The keywords identifying the two loops are:
 - `for` — located at line 4, column 3
 - `while` — located at line 6, column 3
- Immediately after the procedure statistics, a message states that the loop at line 6 in the user program was removed. The compiler recognized that the value of `i` after the first loop is 9999 and that the second loop is not executed.
- The start of the loop at line 4 is marked in the assembly by the ‘Loop at "ExampleC.c" line 4 col 3’ annotation. This annotation follows the loop label `.P34L2` which is used to identify the end of the loop “End Loop `.P34L2`”.

File Position

As seen in Example C (in [“Loop Identification Annotations” on page 2-88](#)), a file position is given, using the file name, line number and the column number in that file as “ExampleC.c” “ line 4 col 5.

Assembly Optimizer Annotations

This scheme uniquely identifies a source code position, unless inlining is involved. In presence of inlining, a piece of code from a certain file position can be inlined at several places, which in turn can be inlined at other places. Since inlining can happen an unspecified number of times, a recursive scheme is used to describe a general file position.

Therefore, a `<general file position>` is `<file position>` inlined from `<general file position>`.

Example D (Inlining Locations)

Consider the following source code:

```
5  void f2(int n);
6  inline void f3(int n)
7  {
8      while(n--)
9          f4();
10         if (n == 7)
11             f2(3*n);
12 }
13
14 inline void f2(int n)
15 {
16     while(n--) {
17         f3(n);
18         f3(2*n);
19     }
20 }
21 void f1(volatile unsigned int i)
22 {
23     f2(30);
24 }
```

Here is some of the code generated for function f1:

```
_f1:
.LN_f1:
//-----
..... procedure statistics .....
```


Achieving Optimal Performance from C/C++ Source Code

```
//Inlined in _f1:
//  ExampleD.c:23:5'_f2
//    ExampleD.c:18:7'_f3
//    ExampleD.c:17:7'_f3
//-----
.....  code .....
.P36L4:
//-----
//    Loop at "ExampleD.c" line 16 col 3 inlined at "ExampleD.c"
line 23 col 5
//-----
.....  loop L4 code .....
.P36L7:
//-----
//    Loop at "ExampleD.c" line 8 col 3 inlined at "ExampleD.c"
line 17 col 7 inlined at "ExampleD.c" line 23 col 5
.....  loop L7 annotations .....
//-----
.....  loop L7 body .....
//-----
//    End Loop L7
//-----
.....  loop L4 code .....
.P36L15:
//-----
//    Loop at "ExampleD.c" line 8 col 3 inlined at "ExampleD.c"
line 18 col 7 inlined at "ExampleD.c" line 23 col 5
.....  loop L15 annotations .....
//-----
.....  loop L15 body .....
//-----
//    End Loop L15
//-----
.....  loop L4 code .....
//-----
//    End Loop L4
//-----
```

Vectorization

The trip count of a loop is the number of times the loop goes around.

Under certain conditions, the compiler is able to take two operations from consecutive iterations of a loop and execute them in a single, more powerful SIMD instruction giving a loop with a smaller trip count. The transformation in which operations from two subsequent iterations are executed in one SIMD operation is called “vectorization”.

For instance, the original loop may start with a trip count of 1000.

```
for(i=0; i< 1000; ++i)
    a[i] = b[i] + c[i];
```

and, after the optimization, end up with the vectorized loop with a final trip count of 250. The vectorization factor is the number of operations in the original loop that are executed at once in the transformed loop. It is illustrated using some pseudo code below.

```
for(i=0; i< 1000; i+=2)
    (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1]);
```

In the above example, the vectorization factor is 2. A loop may be vectorized more than once.

If the trip count is not a multiple of the vectorization factor, some iterations need to be peeled off and executed unvectorized. Thus, if in the previous example, the trip count of the original loop was 1001, then the vectorized code would be:

```
for(i=0; i< 1000; i+=2)
    (a[i], a[i+1]) = (b[i],b[i+1]) .plus2. (c[i], c[i+1]);
a[1000] = b[1000] + c[1000];
    // This is one iteration peeled from
    // the back of the loop.
```

In the above examples the trip count is known and the amount of peeling is also known. If the trip count is not known (it is a variable), the number of peeled iterations depends on the trip count, and in such cases, the optimized code contains peeled iterations that are executed conditionally.

Loop Flattening

Another transformation, related to vectorization, is loop flattening. The loop flattening operation takes two nested loops that run N_1 and N_2 times respectively and transforms them into a single loop that runs $N_1 \cdot N_2$ times. For instance, the following function

```
void copy_v(int a[][100], int b[][100]) {
    int i,j;
    for (i=0; i< 30; ++i)
#pragma SIMD_for
#pragma no_alias
        for (j=0; j < 100; ++j)
            a[i][j] = b[i][j];
}
```

is transformed into

```
void copy_v(int a[][100], int b[][100]) {
    int i,j;
    int *p_a = &a[0][0];
    int *p_b = &b[0][0];
    for (i=0; i< 3000; ++i)
        p_a[i] = p_b[i];
}
```

This may further facilitate the vectorization process:

```
void copy_v(int a[][100], int b[][100]) {
    int i,j;
    int *p_a = &a[0][0];
    int *p_b = &b[0][0];
    for (i=0; i< 3000; i+=2)
        (p_a[i], p_a[i+1]) = (p_b[i], p_b[i+1]);
}
```

Assembly Optimizer Annotations

Example E (Loop Flattening):

The assembly output for the loop flattening example is:

```
_copy_v:
//-----
..... Procedure statistics .....
//-----
// Original Loop at "ExampleE.c" line 6 col 3 -- loop flatten
into loop at "ExampleE.c" line 6 col 5
//-----
..... procedure code .....
.P1L1:
//-----
// Loop at "ExampleE.c" line 6 col 5
..... loop annotations .....
//-----
..... loop body .....
//-----
// End Loop L1
//-----
```

Vectorization Annotations

For every loop that is vectorized, the following information is provided:

- The vectorization factor
- The number of peeled iterations
- The position of the peeled iterations (front or back of the loop)
- Information about whether peeled iterations are conditionally or unconditionally executed

For every loop pair subject to loop flattening, the following information is provided:

- The loop that is lost
- The remaining loop that it was merged with

Example F (Vectorization, for ADSP-21160 Processor):

Consider the test program:

```
void add(int *a, int *b, int* c, int dim) {
    int i;
#pragma no_alias
#pragma SIMD_for
    for (i = 0 ; i < dim; ++i)
        a[i] = b[i] + c[i];
}
```

for which the vectorization information is:

```
bit set model 0x200000; nop;
...
.P34L22:
//-----
//   Loop at "ExampleF.c" line 5 col 3
//-----
//   This loop executes 2 iterations of the original loop in
//   estimated 3 cycles.
//..... other loop annotations .....
//-----
//   Loop was vectorized by a factor of 2.
//-----
//   Vectorization peeled 1 conditional iteration from the
//   back of the loop because of an unknown trip count,
//   possible not a multiple of 2.
//
//   Consider using pragma loop_count to specify the trip count
//   or trip modulo in order to avoid conditional peeling,
//-----
//           r2=r2+r1, r1=dm(i3,2);
//           dm(i4,2)=r2;
//           r2=dm(i5,2);
//           // end loop .P34L22;
//-----
//   End Kernel for Loop L22
//-----
```

Assembly Optimizer Annotations

```
...
    bit clr model 0x200000; nop;
...
```

In this example, the vectorization factor is 2. Since the trip count “dim” is unknown, one conditional iteration is peeled from the back of the loop, corresponding to the case where “dim” is $2k+1$. Note that peeling could be avoided, if additional information about the loop count was provided and the compiler advice “Consider using pragma loop_count to specify the trip count or trip modulo, in order to avoid conditional peeling” informs the user of this.

Modulo Scheduling Information

For every modulo scheduled loop (see also [“Modulo Scheduling” on page 2-64](#)), in addition to regular loop annotations, the following information is provided:

- The initiation interval (II)
- The final trip count if it is known: the trip count of the loop as it ends up in the assembly code
- A cycle count representing the time to run one iteration of the pipelined loop
- The minimum trip count, if it is known and the trip count is unknown
- The maximum trip count, if it is known and the trip count is unknown
- The trip modulo, if it is known and the trip count is unknown
- The stage count (iterations in parallel)
- The MVE unroll factor

- The resource usage
- The minimum initiation interval due to resources (res_MII)
- The minimum initiation interval due to dependency cycles (rec_MII)

Annotations for Modulo Scheduled Instructions

The `-annotate-loop-instr` compiler switch ([on page 1-26](#)) can be used to produce additional annotation information for the instructions that belong to the prolog, kernel or epilog of the modulo scheduled loop.

Consider the example whose schedule is in [Table 2-9 on page 2-73](#). Remember that this does not use a real DSP-architecture, but rather a theoretical one able to schedule four instructions on a line, and each line takes one cycle to execute. We can view the instructions involved in modulo scheduling as in [Table 2-11](#).

Due to variable expansion, the body of the modulo scheduled loop contains $\text{MVE}=2$ unrolled instances of the kernel, and the loop body contains instructions from 4 iterations of the original loop. The iterations in progress in the kernel are shown in the table heading, starting with `Iteration 0` which is the oldest iteration in progress (in its final stage). This example uses two register sets, shown in the table heading.

The instruction annotations contain the following information:

- The part of the modulo scheduled loop (prolog, kernel or epilog)
- The loop label. This is required since prolog and epilog instructions appear outside of the loop body and are subject to being scheduled with other instructions.
- ID: a unique number associated with the original instruction in the unscheduled loop that generates the current instruction. It is useful because a single instruction in the original loop can expand into

Table 2-11. Modulo Scheduled Instructions

	Part	Iteration 0	Iteration 1	Iteration 2	Iteration 3 ...
		Register Set 0	Register Set 1	Register Set 0	Register Set 1
1	prolog	I1			
2	prolog	I2, I3			
3	prolog	I4, I5	I1_2		
4	prolog	I6	I2_2, I3_2		
5		L: Loop ...			
6	kernel	I7	I4_2, I5_2	I1	
7	kernel	I8	I6_2	I2, I3	
8	kernel		I7_2	I4, I5	I1_2
9	kernel		I8_2	I6	I2_2, I3_2
10		END Loop			
11	epilog			I7	I4, I5
12	epilog			I8	I6
13	epilog				I7
14	epilog				I8

multiple instructions in a modulo scheduled loop. In our example the annotations for all instances of I1 and I1_2 have the same id, meaning they all originate from the same instruction (I1) in the unscheduled loop.

The IDs are assigned in the order the instructions appear in the kernel and they might repeat for MVE unroll > 1.

- Loop-carry path, if any. If an instruction belongs to the loop-carry path, its annotation will contain a '*'. If several such paths exist, '*2' is used for the second one, '*3' for the third one, etc.

Achieving Optimal Performance from C/C++ Source Code

- `sn`: the stage count the instruction belongs to.
- `rs`: the register set used for the current instruction (useful when `MVE unroll > 1`, in which case `rs` can be `0, 1, \dots, mve-1`). If the loop has an MVE of 1, the instruction's `rs` is not shown.
- In addition to the above, the instructions in the kernel are annotated with:
 - Iteration. `Iter`: specifies the iteration of the original loop an instruction is on in the schedule.
 - In a modulo scheduled kernel, there are instructions from $(SC+MVE-1)$ iterations of the original loop. `Iter=0` denotes instructions from the earliest iteration of the original loop, with higher numbers denoting later iterations.

Thus, the instructions corresponding to the schedule in [Table 2-11](#) for a hypothetical machine are annotated as follows:

```
1 : I1;          // {L10 prolog:id=1,sn=0,rs=0}
2 : I2,          // {L10 prolog:id=2,sn=0,rs=0}
3 :      I3;      // {L10 prolog:id=3,sn=0,rs=0}
4 : I4,          // {L10 prolog:id=4,sn=1,rs=0}
5 :      I5,      // {L10 prolog:id=5,sn=1,rs=0}
6 :      I1_2;    // {L10 prolog:id=1,sn=0,rs=1}
7 : I6,          // {L10 prolog:id=6,sn=1,rs=0}
8 :      I2_2,    // {L10 prolog:id=2,sn=0,rs=1}
9 :      I3_2;    // {L10 prolog:id=3,sn=0,rs=1}
10: //-----
11: //   Loop at ...
12: //-----
13: //   This loop executes 2 iterations of the original loop in
    estimated 4 cycles.
14: //-----
15: //   Unknown Trip Count
16: //   Successfully found modulo schedule with:
17: //   Initiation Interval (II)           = 2
```

Assembly Optimizer Annotations

```
18: //      Stage Count (SC)                = 3
19: //      MVE Unroll Factor                = 2
20: //      Minimum initiation interval due to recurrences
      (rec MII)                            = 2
21: //      Minimum initiation interval due to resources
      (res MII)                            = 2.00
22: //-----
23:L10:
23:LOOP (N-2)/2;
25: I7,          // {kernel:id=7,sn=2,rs=0,iter=0}
26:   I4_2,      // {kernel:id=4,sn=1,rs=1,iter=1}
27:   I5_2,      // {kernel:id=5,sn=1,rs=1,iter=1,*}
28:   I1;        // {kernel:id=1,sn=0,rs=0,iter=2}
29: I8,          // {kernel:id=8,sn=2,rs=0,iter=0}
30:   I6_2,      // {kernel:id=6,sn=1,rs=1,iter=1}
31:   I2,        // {kernel:id=2,sn=0,rs=0,iter=2}
32:   I3;        // {kernel:id=3,sn=0,rs=0,iter=2,*}
33: I7_2,        // {kernel:id=7,sn=2,rs=1,iter=1}
34:   I4,        // {kernel:id=4,sn=1,rs=0,iter=2}
35:   I5,        // {kernel:id=5,sn=1,rs=0,iter=2,*}
36:   I1_2;      // {kernel:id=1,sn=0,rs=1,iter=3}
37: I8_2,        // {kernel:id=8,sn=2,rs=1,iter=1}
38:   I6,        // {kernel:id=6,sn=1,rs=0,iter=2}
39:   I2_2,      // {kernel:id=2,sn=0,rs=1,iter=3}
40:   I3_2;      // {kernel:id=3,sn=0,rs=1,iter=3,*}
41:END LOOP
42:
43: I7,          // {L10 epillog:id=7,sn=2,rs=0}
44:   I4_2,      // {L10 epillog:id=4,sn=1,rs=1}
45:   I5_2;      // {L10 epillog:id=5,sn=1,rs=1}
46: I8,          // {L10 epillog:id=8,sn=2,rs=0}
47:   I6_2;      // {L10 epillog:id=6,sn=1,rs=1}
48: I7_2;        // {L10 epillog:id=7,sn=2,rs=1}
49: I8_2;        // {L10 epillog:id=8,sn=2,rs=1}
```

Achieving Optimal Performance from C/C++ Source Code

Lines 10-22 define the kernel information: loop name and modulo schedule parameters: II, stage count, and so on.

Lines 25-40 show the kernel.

Each instruction in the kernel has an annotation between {}, inside a comment following the instruction. If several instructions are executed in parallel, each gets its own annotation.

For instance, line 64 looks like:

```
27:      I5_2,      // {kernel:id=5,sn=1,rs=1,iter=1,*}
```

This annotation indicates:

- That this instruction belongs to the kernel of the loop starting at L10.
- That this and the other three instructions that have ID=5 originate from the same original instruction in the unscheduled loop:

```
5 :      I5,          // {L10 prolog:id=5,sn=1,rs=0}
...
27:      I5_2,        // {kernel:id=5,sn=1,rs=1,iter=1,*}
...
35:      I5,          // {kernel:id=5,sn=1,rs=0,iter=2,*}
...
45:      I5_2;        // {L10 epillog:id=5,sn=1,rs=1}
```

- sn=1 shows that this instruction belongs to stage count 1.
- rs=1 shows that this instruction uses register set 1.
- Iter=1 specifies that this instruction belongs to the second iteration of the original loop (Iter numbers are zero-based).
- The ‘*’ indicates that this is part of a loop carry path for the loop. In the original, unscheduled loop, that path is I5 -> I3 -> I5. Due to unrolling, in the scheduled loop the “unrolled” path is I5_2 -> I3->I5->I3_2->I5_2.

Assembly Optimizer Annotations

The prolog and epilog are not clearly delimited in blocks by themselves, but their corresponding instructions are annotated like the ones in the kernel except that they do not have an `Iter` field and that they are preceded by a tag specifying to which loop prolog or epilog they belong:

```
5 :      I5,          // {L10 prolog:id=5,sn=1,rs=0}
...
27:      I5_2,        // {kernel:id=5,sn=1,rs=1,iter=1,*}
...
35:      I5,          // {kernel:id=5,sn=1,rs=0,iter=2,*}
...
45:      I5_2;        // {L10 epilog:id=5,sn=1,rs=1}
```

Note that the prolog/epilog instructions may mix with other instructions on the same line.

This situation does not occur in this example; however, in a different example it might have:

```
      I5_2,          // {L10 epilog:id=5,sn=1,rs=1}
      I20;
```

This shows a line with two instructions. The second instruction `I20` is unrelated to modulo scheduling, and therefore it has no annotation.

Warnings, Failure Messages and Advice

There are innocuous programming constructs that have a negative effect on performance. Since you may not be aware of the hidden problems, the compiler annotations try to give warnings when such situations occur. Also, if a program construct keeps the compiler from performing a certain optimization, the compiler gives the reason why that optimization was precluded.

In some cases, the compiler assumes it could do a better job if you would change your code in certain ways. In these cases, the compiler offers advice on the potentially beneficial code changes. However, take this cautiously. While it is likely that making the suggested change will improve the performance, there is no guarantee that it will actually do so.

Some of the messages are:

- **This loop was not modulo scheduled because it was optimized for space**

When a loop is modulo scheduled, it often produces code that has to precede the scheduled loop (the prolog) and follow the scheduled loop (the epilog). This almost always increases the size of the code. That is why, if you specify an optimization that minimizes the space requirements, the compiler doesn't attempt modulo scheduling of a loop.

- **This loop was not modulo scheduled because it contains calls or volatile operations**

Due to the restrictions imposed by calls and volatile memory accesses, the compiler does not try to modulo schedule loops containing such instructions.

- **This loop was not modulo scheduled because it contains too many instructions**
The compiler does not try to modulo schedule loops that contain many instructions, because the potential for gain is not worth the increased compilation time.
- **This loop was not modulo scheduled because it contains jump instructions**
Only single block loops are modulo scheduled. You can attempt to restructure your code and use single block loops.
- **This loop would vectorize more if alignment were known**
The loop was vectorized, but it could be vectorized even more if the compiler could deduce a stronger alignment of some memory locations used in the loop.
- **This loop would vectorize if alignment were known**
The loop was not vectorized because of unknown pointer alignment.
- **Consider using pragma loop_count to specify the trip count or trip modulo**
This information may help vectorization.
- **Consider using pragma loop_count to specify the trip count or trip modulo, in order to prevent peeling**
When a loop is vectorized, but the trip count is not known, some iterations are peeled from the loop and executed conditionally (based on the run-time value of the trip count). This can be avoided if the trip count is known to be divisible by the number of iterations executed in parallel as a result of vectorization.
- ***operation of this size is implemented as a library call***
This message is issued when a source code operation results in a library call, due to lack of hardware support for performing that operation on operands of that size.

- ***operation* is implemented as a library call**
This message is issued when a source code operation results in a library call, due to lack of direct hardware support. For instance, an integer division results in a library call.
- **MIN operation could not be generated because of unsigned operands**
This message is issued when the compiler detects a MIN operation performed between unsigned values. Such an operation cannot be implemented using the hardware MIN instruction, which requires signed values.
- **MAX operation could not be generated because of unsigned operands**
This message is issued when the compiler detects a MAX operation performed between unsigned values. Such an operation cannot be implemented using the hardware MAX instruction, which requires signed values.
- **Use of volatile in loops precludes optimizations**
In general, volatile variables hinder optimizations. They cannot be promoted to registers, because each access to a volatile variable requires accessing the corresponding memory location. The negative effect on performance is amplified if volatile variables are used inside loops. However, there are legitimate cases when you have to use a volatile variable exactly because of this special treatment by the optimizer. One example would be a loop polling if a certain asynchronous condition occurs. This message does not discourage the use of volatile variables, it just stresses the implications of such a decision.
- **Jumps out of this loop prevent efficient hardware loop generation**
Due to the presence of jumps out of a loop, the compiler either cannot generate a hardware loop, or was forced to generate one that has a conditional exit.

- **Consider using a 4-byte integral type for the variable name, for more efficient hardware loop generation**
Using short-typed variables as loop control variables limits optimization because the short variables may wrap. For instance, in the following example,

```
unsigned short i;  
for (i = 0; i < c; i++)  
    ....
```

if $c > 65536$, then the loop will run forever because i wraps from 65535 back to 0. In this case, the compiler must add a wrapper. The compiler recommends using an `int` variable instead (`int` or `unsigned int`) unless the smaller size is critical to your program's behavior.

- **There are N more instructions related to this call**
Certain operations are implemented as library calls. In those cases the call instruction in the assembly code is annotated explaining that the user operation was implemented as a call. However the cost of the operation may be slightly larger than the cost of the call itself, due to additional overhead required to pass the parameters and to obtain the result. This message gives an estimate of the number of instructions in such an overhead associated with a library call.
- **This function calls the “alloca” function which may increase the frame size**
The assembly annotations try to estimate the frame size for a given function. However, if the function makes explicit use of `alloca` then this increases the frame size beyond the original reported estimate.

I INDEX

Numerics

128-bit alignment, [1-159](#)

__2106x__ macro, [1-259](#), [1-260](#)

__2116x__ macro, [1-259](#)

__2126x__ macro, [1-259](#), [1-260](#)

__2136x__ macro, [1-259](#), [1-261](#)

__2137x__ macro, [1-259](#), [1-261](#), [1-262](#)

__213xx__ macro, [1-259](#), [1-261](#)

__2146x__ macro, [1-259](#)

2146x processors

generating normal word size, [1-49](#)

generating short word size, [1-62](#), [1-63](#)

__214xx__ macro, [1-259](#)

32-bit alignment, [1-159](#)

32-bit floating-point arithmetic, [1-80](#)

32-bit IEEE single-precision format, [1-30](#)

64-bit alignment, [1-159](#)

64-bit floating-point arithmetic, [1-80](#)

A

-A (assert) compiler switch, [1-23](#)

action qualifier keywords, for use with #pragma
diag, [1-208](#)

-add-debug-libpaths compiler switch, [1-24](#)

__ADI_LIBEH__ macro, [1-32](#)

__ADI_THREADS macro, [1-64](#)

__ADSP21000__ macro, [1-57](#), [1-260](#), [1-261](#)

__ADSP21020__ macro, [1-260](#)

ADSP-21020 processor, stack frame, [1-299](#)

__ADSP21060__ macro, [1-260](#)

__ADSP21061__ macro, [1-260](#)

__ADSP21062__ macro, [1-260](#)

__ADSP21065L__ macro, [1-260](#)

ADSP-2106x/2116x/2126x/2136x processors,
stack frame, [1-301](#)

ADSP-2106x processors, data corruption in use
code, [1-252](#)

ADSP-210xx/2116x/2126x processors, stack
and heap memory allocation, [1-277](#)

__ADSP21160__ macro, [1-260](#)

ADSP-21160M processor, anomaly #40, [1-239](#)

__ADSP21161__ macro, [1-260](#)

ADSP-21161 processor

anomaly #45, [1-87](#)

executing code from external SDRAM, [1-217](#)

__ADSP21261__ macro, [1-260](#)

__ADSP21262__ macro, [1-260](#)

__ADSP21266__ macro, [1-260](#)

__ADSP21267__ macro, [1-260](#)

ADSP-2126x/2136x processors

data placement, [1-246](#)

data transfer between internal and external
memory, [1-247](#)

__ADSP21363__ macro, [1-261](#)

__ADSP21364__ macro, [1-261](#)

__ADSP21365__ macro, [1-261](#)

__ADSP21366__ macro, [1-261](#)

__ADSP21367__ macro, [1-261](#)

__ADSP21368__ macro, [1-261](#)

__ADSP21369__ macro, [1-261](#)

__ADSP21371__ macro, [1-261](#)

__ADSP21375__ macro, [1-261](#)

ADSP-21375 memory map change, [1-257](#)

INDEX

- ADSP-213xx processors, stack and heap
 - memory allocation, [1-277](#)
- `__ADSP21462__` macro, [1-261](#)
- `__ADSP21465__` macro, [1-261](#)
- `__ADSP21467__` macro, [1-261](#)
- `__ADSP21469__` macro, [1-262](#)
- aggregate assignment support (compiler), [1-144](#)
- aggregate constructor expression, [1-144](#)
- alias, avoiding, [2-21](#)
- aligned-stack compiler switch, [1-25](#)
- alignment inquiry keyword, [1-226](#)
- `__alignof__` (type-name) construct, [1-226](#)
- alldata section identifier, [1-61](#), [1-138](#)
- alter macro, [1-318](#)
- alternate
 - heaps, accessed with standard interface, [1-289](#)
 - keywords, [1-45](#)
 - registers, [1-293](#), [1-297](#)
- alternate heap interface functions
 - C++ run-time support for, [1-291](#)
 - entry point names, [1-290](#)
 - list of, [1-290](#)
- alternative
 - operator keywords, [1-25](#)
 - tokens, disabling, [1-25](#)
 - tokens, enabling, [1-25](#)
 - tokens in C, [1-25](#)
- alttok (alternative tokens) compiler switch, [1-25](#)
- ALU saturation, disabling, [1-296](#)
- always-inline compiler switch, [1-26](#), [1-109](#)
- anach (enable C++ anachronisms) C++
 - mode compiler switch, [1-73](#)
- anachronisms
 - default C++ mode, [1-73](#)
 - disabling in C++ mode, [1-76](#)
- `__ANALOG_EXTENSIONS__` macro, [1-262](#)
- annotate (enable assembly annotations)
 - compiler switch, [1-26](#)
- annotate-loop-instr compiler switch, [1-26](#), [2-89](#)
- annotation information, instrumental, [1-26](#)
- annotations
 - assembly code, [2-80](#)
 - assembly source code position, [2-92](#)
 - disabling, [1-26](#), [1-42](#)
 - embedded, [2-7](#)
 - enabling, [1-60](#)
 - loop identification, [2-87](#)
 - modulo scheduling, information
 - provided, [2-98](#)
 - modulo scheduling, parameters, [2-64](#)
 - source and assembly, [2-7](#)
 - vectorization, [2-96](#)
- anomalies
 - #40 (SIMD), [1-240](#)
 - IDs, [1-87](#)
 - workaround management, [1-85](#)
 - workaround switch, [1-88](#)
- ANSI standard
 - compiler, [1-33](#)
- archiver, [1-3](#)
- argc
 - support, [1-282](#)
- arguments and return transfer, [1-304](#)
- argv
 - support, [1-282](#)
- argv/argc arguments, [1-282](#)
- `__argv_string` variable, defining, [1-283](#)
- array
 - initializer, [1-142](#)
 - storage, [1-307](#)
 - zero length, [1-224](#)

- asm
 - compiler keyword, [1-106](#), [1-113](#)
 - construct template operands, [1-118](#)
 - keyword, [1-113](#), [1-226](#)
 - statement, [1-224](#), [2-25](#)
 - workarounds not applied, [1-85](#), [1-113](#)
- asm() construct
 - described, [1-113](#), [1-126](#)
 - flow control, [1-128](#)
 - input operand, [1-116](#)
 - optimizing, [1-125](#)
 - reordering, [1-125](#)
 - syntax, elements, [1-115](#)
 - syntax, rules, [1-116](#)
 - template, [1-115](#)
 - with compile-time constant, [1-127](#)
 - with multiple instructions in atemplate, [1-124](#)
- asm() operand constraints, [1-122](#)
- asm_sprt.h system header file, [1-316](#)
- asm() statement, using, [1-129](#)
- asm volatile() construct, [1-126](#)
- assembler
 - for SHARC processors, [1-3](#)
- assembly
 - annotations, [2-7](#)
 - code annotations, [2-80](#)
 - instruction operands, [1-115](#)
 - support keyword (asm), [1-330](#)
- assembly construct
 - flow control, [1-128](#)
 - operand description, [1-118](#)
 - reordering and optimization, [1-125](#)
 - syntax, [1-115](#)
 - with multiple instructions, [1-124](#)
- assembly language support keyword (asm)
 - constructs with multiple instructions, [1-124](#)
- assembly optimizer
 - annotations, [2-80](#)
 - file position, [2-92](#)
 - global information, [2-81](#)
 - loop flattening, [2-95](#)
 - loop identification annotation, [2-88](#)
 - messages and warnings, [2-105](#)
 - modulo scheduling, [2-64](#), [2-98](#)
 - procedure statistics, [2-82](#)
 - vectorization, annotations, [2-96](#)
 - vectorization, example, [2-94](#)
- assembly output annotations
 - disabling, [1-26](#), [1-42](#)
 - disabling via IDDE, [1-26](#), [1-43](#)
 - enabling annotations, [1-26](#)
 - enabling via IDDE, [1-26](#), [1-43](#)
 - failure messages, [2-105](#)
 - file position, [2-91](#)
 - global information, [2-81](#)
 - in saved assembly file, [2-80](#)
 - loop flattening, [2-95](#)
 - loop identification, [2-87](#)
 - modulo scheduling, [2-64](#), [2-98](#)
 - of generated source code, [2-7](#)
 - procedure statistics, [2-82](#)
 - selecting, [2-80](#)
 - vectorization, defined, [2-94](#)
 - warnings, [2-105](#)
- assembly routine
 - exceptions table in, [1-340](#)
- assembly subroutine, calling from C/C++
 - program, [1-312](#)
- atexit() library routine, [1-281](#)
- __attribute__ keyword, [1-227](#)

INDEX

- attributes
 - adding to a file, [1-353](#)
 - file, [1-27](#), [1-33](#), [1-43](#), [1-348](#)
 - functions, variables and types, [1-227](#)
 - names, [1-348](#)
 - usage examples, [1-353](#)
 - value, [1-348](#)
- auto-attrs compiler switch, [1-27](#)
- autoinit section identifier, [1-61](#), [1-138](#)
- automatic
 - attributes, disabling, [1-43](#)
 - attributes, enabling, [1-27](#)
 - function inlining, [1-49](#)
 - inlining, [1-82](#), [1-108](#), [2-24](#)
 - inlining, controlled with the - Ov num switch, [1-52](#)
 - loop control variables, [2-36](#)
 - variables, [1-130](#)
- automatically-applied attributes, [1-348](#)
- automatic attributes
 - disabling, [1-43](#)
 - enabling, [1-27](#)

B

- background registers, [1-293](#), [1-297](#)
- bank qualifier, [1-135](#)
- binary object granularity, [1-352](#)
- bit-fields
 - signed, [1-62](#)
 - unsigned, [1-65](#)
 - values, [1-66](#)

- Blackfin-specific functionality
 - argv/argc arguments, [1-282](#)
- bool, *See* Boolean type support keywords (bool, true, false)
- Boolean type support keywords (bool, true, false), [1-139](#)
- boot loader, [1-274](#)
- bsz section identifier, [1-61](#), [1-138](#)
- build-lib (build library) compiler switch, [1-27](#)
- build tools, [1-34](#)
- __builtin_aligned function, [2-14](#), [2-19](#), [2-52](#)
- __builtin_assert() function, [1-153](#)
- __builtin_circindex function, [2-42](#)
- __builtin_circptr function, [2-42](#)
- built-in functions
 - circular buffer, [1-151](#)
 - defined, [1-146](#)
 - expected_false, [1-152](#)
 - expected_true, [1-152](#)
 - funcsize, [1-156](#)
 - ignoring, [1-43](#)
 - in code optimization, [2-39](#)
 - system support, [2-39](#)

C

- C
 - tokens in, [1-25](#)
 - variable-length arrays, [1-140](#)

- C++
 - alternative tokens in, 1-25
 - class constructor functions, 1-61, 1-138
 - class instance function, 1-306
 - compiler switches, 1-73
 - constructors and destructors, 1-280
 - exceptions, 1-218
 - fractional arithmetic, 1-228
 - gcc compatibility features not supported, 1-219
 - language extension, fract data type, 1-107
 - member functions in assembly language, 1-322
 - programming examples, complex support, 1-327
 - programming examples, fract support, 1-326
 - programming examples, running tips, 1-325
 - run-time libraries rationalization, 1-257
 - style comments, 1-145
 - support tables (ctor, gdt), 1-254
 - template inclusion control pragma, 1-190
 - templates, 1-344
 - virtual lookup tables, 1-61, 1-138
- c89 (ISO/IEC 9899
 - 1990 standard) compiler switch, 1-22
- calling
 - assembly language subroutine, 1-312
 - assembly language subroutines from C/C++ programs, 1-312
 - C/C++ functions from assembly language programs, 1-314
- calloc heap function, 1-283
- call preserved registers, 1-295
- call preserved registers (pass array), 1-338
- C++ anachronisms
 - disabling, 1-76
 - enabling, 1-73
- C/C++
 - callable subroutines in SIMD mode, 1-324
 - code optimization, 2-2
 - data types, 1-78
 - functions, calling from assembly program, 1-314
 - SIMD mode constraints in, 1-238
 - switch statements, 1-61, 1-138
- cc21k compiler
 - See also* compiler
 - defined, 1-1
 - overview, 1-3
 - running from command line, 1-8
- ccall macro, 1-318, 1-330
- C/C++ assembly interface, *See* mixed C/C++ assembly programming
- C/C++ compiler, overview, 1-3
- C/C++ language extensions
 - aggregate assignments, 1-107
 - asm keyword, 1-113
 - bool keyword, 1-106
 - dm keyword, 1-130
 - false keyword, 1-106
 - indexed initializers, 1-107
 - inline, 1-108
 - inline keyword, 1-108
 - long identifiers, 1-107
 - non-constant initializers, 1-107
 - pm keyword, 1-130
 - section keyword, 1-106
 - table describing, 1-106
 - true keyword, 1-106
 - variable length arrays, 1-107
- c++ (C++ mode) compiler switch, 1-22

INDEX

- C/C++ mode selection switches
 - c89, [1-22](#)
 - c++ (C++ mode), [1-22](#)
- C (comments) compiler switch, [1-27](#)
- c (compile only) compiler switch, [1-27](#)
- C compiler
 - benchmarking performance of, [1-279](#)
 - overview, [1-91](#)
 - switches, [1-71](#)
- C/C++ run-time environment, [1-267](#)
 - See also* mixed C/C++/assembly programming
- char storage, [1-307](#)
- check-init-order C++ mode compiler switch, [1-74](#), [1-281](#)
- circular buffer code, disabling automatic generation of, [1-44](#)
- circular buffers
 - __builtin_circindex function, [2-42](#)
 - __builtin_circptr function, [2-42](#)
 - built-in functions, [1-151](#)
 - enabling by setting CBUFEN, [1-296](#)
 - enabling for use, [1-34](#)
 - increment of index, [1-151](#)
 - increment of pointer, [1-151](#)
 - increments for modulus array references, [1-151](#)
 - interrupt dispatchers, described, [1-248](#)
 - interrupt dispatchers, saving data, index, modify, length registers, [1-248](#)
 - interrupt dispatchers, zeroing Length registers, [1-251](#)
 - used in DSP-style code, [2-40](#)
 - used with the -force-circbuf compiler switch, [2-41](#)
- cjump instruction, [1-301](#)
- C language extensions
 - C++ style comments, [1-107](#)
 - preprocessor generated warnings, [1-107](#)
- class conversion optimization pragmas, [1-184](#)
- classes
 - initializing global instances, [1-280](#)
- class pointers, converting, [1-184](#)
- clobber, of asm() construct, [1-116](#)
- clobbered
 - register definition, [2-55](#)
 - registers, [1-176](#), [1-177](#), [1-178](#)
- C++ mode, compiling in, [1-22](#)
- C mode compiler switches
 - misra, [1-71](#)
 - misra-linkdir, [1-71](#)
 - misra-no-cross-module, [1-71](#)
 - misra-no-runtime, [1-71](#)
 - misra-strict, [1-72](#)
 - misra-suppress-advisory, [1-72](#)
 - misra-suppress-testing, [1-72](#)
 - Wmis_suppress, [1-72](#)
 - Wmis_warn_rule_number, [1-73](#)
- C++ mode compiler switches
 - anach (enable C++ anachronisms), [1-73](#)
 - check-init-order, [1-74](#), [1-281](#)
 - eh (enable exception handling), [1-32](#)
 - full-dependency-inclusion, [1-75](#)
 - ignore-std, [1-75](#)
 - no-anach (disable C++ anachronisms), [1-76](#)
 - no-eh (disable exception handling), [1-45](#)
 - no-implicit-inclusion, [1-76](#)
 - no-rtti (disable run-time type identification), [1-76](#)
 - no-std-templates, [1-76](#)
 - rtti (enable run-time type identification), [1-76](#)
 - std-templates, [1-77](#)
- code generation pragmas, [1-217](#)
- code inlining, controlling, [1-192](#)
- CODE memory area, [1-282](#)

- code optimization
 - built-in functions, [2-39](#)
 - controlling, [2-4](#)
 - enabling, [1-49](#)
 - for maximum performance, [2-44](#)
 - for size, [1-50, 2-43](#)
 - for speed, [1-50](#)
 - using function pragmas, [2-45](#)
 - using loop optimization pragmas, [2-50](#)
 - using pragmas for, [2-45](#)
 - using pragmas in, [2-45](#)
 - with PGO, [2-8](#)
- code section identifier, [1-61, 1-138](#)
- command-line
 - interface, [1-7](#)
 - syntax, [1-8](#)
- comma-separated section qualifiers, [1-204](#)
- compatible-pm-dm compiler switch, [1-27](#)
- compilation time, indicating with the
 - no-progress-req-timeout compiler switch, [1-46](#)

- compiler
 - building for a specific hardware revision, [1-62, 1-86](#)
 - built-in functions, [1-146](#)
 - C/C++ extensions, [1-104, 1-106](#)
 - code generator workarounds, [1-88](#)
 - code optimization, [1-80, 2-2](#)
 - command-line interface, overview, [1-7](#)
 - command-line switch summaries, [1-9](#)
 - command-line syntax, [1-8](#)
 - diagnostic messages, [1-207](#)
 - diagnostics, [2-5](#)
 - disabling GNU compatibility mode, [1-46](#)
 - disabling hardware anomaly workarounds, [1-48](#)
 - enabling GNU compatibility mode, [1-41](#)
 - enabling hardware anomaly workarounds, [1-69](#)
 - enabling hardware anomaly workarounds, [1-88](#)
 - generating a label, [1-114](#)
 - keywords, not recognized, [1-45](#)
 - optimizer, [2-4](#)
 - overview, [1-3](#)
 - prelinker, [1-84](#)
 - producing processor-specified code, [1-57](#)
 - progress feedback, [1-57, 1-58](#)
 - registers, [1-293](#)
 - selecting specified compilation tool, [1-54](#)
 - starting a new optimization pass, [1-58](#)
 - stopping after compilation, [1-60](#)
 - undefining macros, [1-65](#)
- compiler driver, [1-87](#)
- compile-time constant, [1-127](#)
- compound macros, [1-265](#)
- compound statement., [1-265](#)
- conditional code
 - avoiding in loops, [2-34](#)
 - improving, [2-28](#)

INDEX

- conditional expressions, with missing operands, [1-223](#)
- const, pointers, [1-28](#)
- constants
 - accessed as read-write data, [1-28](#)
 - initializing statically, [2-17](#)
- constdata section identifier, [1-61](#), [1-138](#)
- constraint, of asm() construct, [1-115](#)
- const-read-write compiler switch, [1-28](#)
- constructors
 - C++ classes, [1-280](#), [1-281](#)
- constructors and destructors
 - and memory placement, [1-281](#)
 - for global class instances, [1-280](#)
 - start routine, [1-280](#)
- constructs
 - flow control, [1-128](#)
 - input and output operands, [1-126](#)
 - with compile-time constant, [1-127](#)
- const-string compiler switch, [1-28](#)
- Content attributes, to map binary objects, [1-349](#), [1-350](#)
- continuation characters, [1-41](#), [1-46](#)
- controlling code inlining, [1-192](#)
- core algorithm, unmodified, [2-11](#)
- count_ticks() function, [1-214](#)
- __cplusplus macro, [1-262](#)
- cross-reference listing information, [1-70](#)
- C++ STL objects, [1-286](#)
- ctdm memory section, [1-281](#)
- __ctor_loop function, [1-281](#)
- custom allocator, [1-286](#)
- customer support, [-xxxii](#)

D

- DAG registers, [1-88](#), [1-122](#)
- data
 - alignment pragmas, [1-158](#)
 - dual-word-aligned, [2-18](#)
 - fetching with 32-bit loads, [2-18](#)
 - memory storage, [1-272](#)
 - storage formats, [1-307](#)
 - transfer between internal and external memory, [1-247](#)
 - word alignment, [2-19](#)
- DATA memory area, [1-282](#)
- data placement
 - compiler-controlled, [1-60](#)
 - controlled by the -section id compiler switch, [1-137](#)
 - link-time checking of, [1-246](#)
- data section identifier, [1-61](#), [1-138](#)
- data storage initialization, [1-274](#)
- data types
 - bit sizes, [1-78](#)
 - double, [1-80](#)
 - float, [1-79](#), [1-80](#)
 - fract, [1-79](#), [1-228](#)
 - int, [1-79](#)
 - long, [1-79](#)
 - long double, [1-80](#)
 - long int, [1-79](#)
 - scalar, [2-15](#)
- __DATE__ macro, [1-262](#)
- D (define macro) compiler switch, [1-28](#), [1-65](#)
- debugger, generating debug line information, [1-114](#)
- debugging, source-level, [1-35](#)

- debugging information
 - debug optimization level, [1-81](#)
 - for header file, [1-29](#)
 - generating, [1-35](#)
 - lightweight, [1-36](#)
 - preserving, [1-50](#)
 - removing, [1-60](#)
 - with the `-g` switch, [1-35](#)
 - Debug subdirectory, [1-24](#)
 - `-debug-types` compiler switch, [1-29](#)
 - declarations, mixed with code, [1-225](#)
 - default
 - LDF placement, [1-351](#)
 - names, controlling, [1-60](#), [1-137](#)
 - preprocessor macros, disabling, [1-44](#)
 - run-time header file, [1-275](#)
 - sections, [1-201](#)
 - target processor, [1-57](#)
 - `default_section` pragma, [1-137](#), [1-138](#)
 - `#define` preprocessor command, [1-258](#), [1-264](#)
 - definition, unique identifier to, [1-196](#)
 - delayed branches, disabled, [1-44](#)
 - delete operator, with multiple heaps, [1-291](#)
 - dependent name processing
 - disabling, [1-76](#)
 - enabling, [1-77](#)
 - destructors
 - C++ classes, [1-280](#), [1-281](#)
 - diagnostic messages
 - modifying behavior, [1-209](#)
 - restoring behavior, [1-209](#)
 - saving behavior, [1-209](#)
 - severity of, [1-207](#)
 - diagnostics
 - annotations, [2-7](#)
 - control pragma, [1-207](#)
 - described, [2-5](#)
 - remarks, [2-6](#)
 - warnings, [2-6](#)
 - diagnostic warnings, enabling, [1-68](#)
 - `dm`, *See* dual memory support keywords (pm, dm)
 - `dm`, *See* dual memory support keywords (pm,dm)
 - `dmaonly` keyword, [1-208](#)
 - DMAONLY qualifier, [1-246](#)
 - DM qualifier, [1-204](#)
 - double
 - 32-bit data type, [1-29](#)
 - 64-bit data type, [1-29](#)
 - data type formats, [1-29](#)
 - storage format, [1-308](#)
 - DOUBLE32 qualifier, [1-204](#)
 - DOUBLE64 qualifier, [1-204](#)
 - DOUBLEANY qualifier, [1-204](#)
 - `__DOUBLES_ARE_FLOATS__` macro, [1-30](#), [1-262](#)
 - `-double-size-32` (single-precision double) compiler switch, [1-29](#)
 - `-double-size-64` (double-precision double) compiler switch, [1-29](#)
 - `-double-size-any` compiler switch, [1-30](#)
 - driver I/O
 - pipe, enabling, [1-69](#)
 - redirection, enabling, [1-69](#)
 - `-dry-run` (verbose dry-run) compiler switch, [1-31](#)
 - `-dry` (terse `-dry-run`) compiler switch, [1-31](#)
 - dual compute-block architectures, [2-18](#), [2-19](#)
 - dual-memory support keywords (pm dm), [1-130](#)
 - dual-word-aligned addresses, [2-18](#)
 - dual-word boundary, [2-20](#)
 - `dynamic_cast` run-time type identification, [1-76](#)
- E
- `iasm21k` assembler, [1-3](#)

INDEX

`__ECC__` macro, [1-262](#)
`__EDG__` macro, [1-262](#)
`__EDG_VERSION__` macro, [1-262](#)
`-ED` (run after preprocessing to file)
 compiler switch, [1-31](#)
`-EE` (run after preprocessing) compiler
 switch, [1-31](#)
`-eh` (enable exception handling) C++ mode
 compiler switch, [1-32](#)
elfar archive library, [1-3](#)
elfloader utility, [1-274](#)
emulated arithmetic, avoiding, [2-16](#)
entry macro, [1-301](#), [1-317](#)
enumeration types, [1-33](#)
`-enum-is-int` compiler switch, [1-33](#)
environment variables
 ADI_DSP, [1-78](#)
 CC21K_IGNORE_ENV, [1-78](#)
 CC21K_OPTIONS, [1-78](#)
 PATH, [1-77](#)
 TEMP, [1-77](#)
 TMP, [1-77](#)
errata workarounds, [1-86](#)
error keyword, [1-208](#)
error messages
 control pragma, [1-207](#)
 overriding, [1-67](#)
escape character, [1-226](#)
`-E` (stop after preprocessing) compiler
 switch, [1-31](#)
exception handler
 disabling, [1-45](#)
 enabling, [1-32](#)
`__EXCEPTIONS` macro, [1-32](#), [1-262](#)
exceptions table, [1-218](#), [1-340](#)
`exit()` library routine, [1-281](#)
exit macro, [1-302](#), [1-317](#)
`expected_false` built-in function, [1-152](#),
 [2-28](#)

`expected_true` built-in function, [1-152](#),
 [2-28](#)
external memory
 accessing from processor core, [1-246](#)
 accessing with inline functions, [1-247](#)
 SIMD access to, [1-236](#)
 using the `dmaonly` keyword with, [1-208](#)
`.EXTERN` assembler directive, [1-321](#)
`-extra-keywords` (not quite `-analog`)
 compiler switch, [1-33](#)

F

false, *See* Boolean type support keywords
 (bool, true, false)
faster operations, disabling, [1-46](#)
fast hardware floating-point instructions,
 [1-30](#)
fast interrupt dispatcher
 described, [1-249](#)
 saving scratch registers, [1-249](#)
file
 annotation position, [2-91](#)
 attributes, [1-348](#)
 attributes, adding, [1-33](#)
 attributes, automatically-applied, [1-348](#)
 attributes, disabling, [1-43](#)
 automatic attributes, [1-27](#)
 extensions, [1-8](#)
 multiple attributes, [1-33](#)
`-file-attr` (file attribute) compiler switch,
 [1-33](#)
`__FILE__` macro, [1-262](#)
file name
 reading from, [1-23](#)
 to be processed, [1-23](#)
`-@ filename` (command file) compiler
 switch, [1-23](#)
file name (description), [1-23](#)
file-to-device stream, [1-82](#)
FIX instruction, [1-296](#)

- flags (command line input) compiler switch, [1-34](#)
- FLAGS registers, [1-164](#)
- floating-point
 - data types, [1-79](#)
 - hexadecimal constants, [1-223](#)
 - underflow, avoiding, [1-34](#)
- floating-point multiplication and addition
 - as associative operations, [1-35](#)
 - not as associative operations, [1-45](#)
- float storage format, [1-29](#), [1-308](#)
- float-to-int compiler switch, [1-34](#)
- float to integer conversion, [1-34](#)
- flow control operations, [1-128](#)
- force-circbuf (circular buffer) compiler switch, [1-34](#), [2-41](#)
- FORCE_CONTIGUITY linker directive, [1-255](#)
- fp-associative (floating-point associative operation) compiler switch, [1-35](#)
- fract data type, [1-79](#), [1-228](#)
- fractional
 - arithmetic in C++ mode, [1-228](#)
 - arithmetic operators, [1-229](#)
 - literals, [1-229](#)
 - saturated arithmetic, [1-230](#)
- frame pointer, [1-298](#)
- free heap function, [1-283](#)
- full-dependency-inclusion C++ mode compiler switch, [1-75](#)
- full-version (display versions) compiler switch, [1-35](#)
- FuncName attributes, [1-350](#)
- functsize built-in function, [1-156](#)

- function
 - always-inline switch, [1-26](#)
 - arguments/return value transfer, [1-304](#)
 - calling in loop, [2-35](#)
 - call return address, [1-330](#)
 - declarations with pointers, [1-133](#)
 - entry (prologue), [1-298](#), [1-330](#)
 - exit (epilogue), [1-298](#), [1-330](#)
 - inlining, about, [1-108](#)
 - out-of-line copy, [1-111](#)
- function call, [2-35](#)
- function call, reported statistics for, [2-82](#)
- function inlining
 - global asm statements, [1-112](#)
 - how to use, [2-24](#)
 - ignoring section directives, [1-112](#)
 - optimization, [1-111](#)
 - out-of-line copies, [1-111](#)
- functions
 - functsize, [1-156](#)
 - obtaining size in bits, [1-156](#)
- function side-effect pragmas
 - for code optimization, [2-45](#)
 - listed with example, [1-173](#)

G

- GCC compatibility extensions, [1-219](#)
- general optimization pragmas, [1-171](#)
- gets macro, [1-318](#)
- g (generate debug information) compiler switch, [1-35](#)
- glite (lightweight debugging) compiler switch, [1-36](#)
- global asm statements, and inlining, [1-112](#)
- .GLOBAL assembler directive, [1-321](#)
- global data, [1-272](#)
- global information, [2-81](#)
- global variable debugging, [1-35](#)
- globvar global variable, [2-36](#)
- GNU C compiler, [1-219](#)

INDEX

GNU compatibility mode

 disabling, [1-46](#)

 enabling, [1-41](#)

granularity, [1-352](#)

guard, [2-50](#)

H

hardware

 defect workarounds, [1-88](#)

 loops, nested, [1-59](#)

 pipelining, [2-59](#)

hardware revision, building project for,

[1-62](#), [1-86](#)

header

 precompiled, [1-55](#)

 stop point, [1-189](#)

header file control pragmas, [1-189](#)

header files

 providing symbolic names for individual
 bits, [1-150](#)

heap

 allocation and initialization example,
 [1-291](#)

 alternate, [1-289](#), [1-290](#)

 C program examples, [1-291](#)

 declaring, [1-284](#)

 identifiers, [1-286](#)

 interface with alternate heaps, [1-291](#)

 memory allocation, [1-277](#)

 standard functions, [1-283](#)

heap_calloc function, [1-283](#), [1-290](#)

heap_free function, [1-283](#), [1-290](#)

heap_lookup_name function, [1-286](#)

heap_malloc function, [1-283](#), [1-290](#)

heap_realloc function, [1-283](#), [1-290](#)

heaps

 non-default, [1-286](#)

heap_switch function, [1-289](#)

-help (command-line help) compiler
switch, [1-36](#)

hexadecimal floating-point constants,
[1-223](#)

-HH (list headers and compile) compiler
switch, [1-36](#)

-H (list headers) compiler switch, [1-36](#)

hoisting, [2-57](#)

__HOSTNAME__ macro, [1-64](#)

I

IDDE_ARGS macro, [1-283](#)

identifier, long, [1-141](#)

IEEE single-/double-precision formats,
[1-307](#)

-ignore-std C++ mode compiler switch,
[1-75](#)

-I (include search directory) compiler
switch, [1-36](#), [1-48](#)

-i (less includes) compiler switch, [1-38](#)

IMASKP interrupt latch register, [1-271](#)

implicit inclusion

 defined, [1-190](#)

 disabling, [1-76](#)

 enabling, [1-75](#)

 of .cpp files, [1-75](#)

implicit instantiation method, [1-344](#)

implicit pointer conversion, [1-38](#)

-implicit-pointers compiler switch, [1-38](#)

#include command, [1-258](#)

include directory list, [1-37](#)

include files, searching, [1-37](#)

-include (include file) compiler switch,
[1-39](#)

incomplete function prototype, [1-68](#)

index, starting value for, [1-151](#)

indexed

 array, [2-22](#)

 initializer support (compiler), [1-142](#)

 style, [2-23](#)

induction variables, [2-33](#)

- initialization
 - data, 1-274
 - data storage, 1-274
 - memory, 1-41
 - order, 1-75
 - section, processing of, 1-274
- initializer
 - indexed, 1-142
 - memory, 1-41
 - non-constant, 1-142
- initiation interval
 - described, 2-64
 - kernel, 2-65
- inline
 - asm statements, 2-25
 - assembly language support keyword
 - (asm), 1-113, 1-115, 1-125
 - automatic, 2-24
 - code, avoiding, 2-44
 - constructs, 1-124
 - control pragmas, 1-192
 - expansion of C/C++ functions, 1-49
 - file position, 2-92
 - function, 2-24
 - function support keyword, example, 1-108
 - function support keyword, inline, 1-106
 - keyword, avoiding use of, 2-44
 - keyword, described, 1-108
 - keyword, using, 2-24
 - qualifier, 1-109, 1-193
- inline assembly (add) example, 1-330
- inline keyword, 1-194
- inline qualifier
 - enabling, 1-26
 - ignoring, 1-42
- inlining, with #pragma inline, 1-109, 1-110, 1-194
- inner loops
 - improving performance of, 2-33
 - producing optimal code for, 2-50
- input operands, 1-116, 1-126
- installation location, 1-54
- instantiation, template functions, 1-187
- integer data types, 1-79
- interface support macros
 - alter, 1-318
 - ccall, 1-318
 - C/C++ and assembly, 1-316
 - entry, 1-317
 - exit, 1-317
 - gets, 1-318
 - leaf_entry, 1-317
 - leaf_exit, 1-317
 - puts, 1-318
 - reads, 1-318
 - restore_reg, 1-319
 - save_reg, 1-318
- interface support macros, described, 1-321
- interfacing C/C++ and assembly, *See* mixed C/C++/assembly programming
- intermediate files, saving, 1-60
- interprocedural analysis (IPA)
 - code optimization with, 1-84
 - defined, 1-84
 - enabling, 1-39, 1-83, 2-13
 - framework, 1-195
 - generating usage information, 1-85
 - identifying variables, 2-17
 - ipa compiler switch for, 1-39, 1-84, 2-13
 - #pragma core used with, 1-195
- interprocedural optimizations
 - described, 1-83
 - when to use, 2-13

INDEX

- interrupt
 - circular buffering, [1-251](#)
 - dispatchers, [1-247](#)
 - handler, use of, [1-163](#)
 - handler pragmas, [1-162](#), [1-165](#)
 - IRPTEN enable bit, [1-252](#)
 - length (in words), [1-277](#)
 - nesting, allowed, [1-248](#), [1-251](#)
 - nesting, disabled, [1-250](#)
 - nesting, restrictions on
 - ADSP-2116x/2126x/2136x chips, [1-252](#)
 - pragmas, [1-165](#)
 - self-modifying code, [1-251](#)
 - set-up functions, [1-251](#)
 - table, [1-310](#)
 - using non-self-modifying function, [1-251](#)
 - vector table, [1-275](#)
 - interruptcb() function, [1-248](#)
 - interrupt dispatchers
 - circular buffer, [1-251](#)
 - described, [1-248](#)
 - fast, [1-249](#)
 - normal, [1-249](#)
 - pragma, [1-250](#), [1-251](#)
 - super-fast, [1-250](#)
 - interruptf() function, [1-249](#)
 - interruptfnsm() function, [1-251](#)
 - interrupt() function, [1-249](#)
 - interrupt service routines, *See* ISRs
 - interrupts() function, [1-250](#), [1-252](#)
 - interruptss() function, [1-250](#)
 - interrupt vector table, [1-165](#)
 - intrinsic (built-in) functions, [1-146](#)
 - int storage format, [1-307](#)
 - IPA, *See* interprocedural analysis (IPA)
 - IPA framework, and #pragma core, [1-195](#)
 - ipa (interprocedural analysis) compiler switch, [1-39](#), [1-84](#), [2-13](#)
 - IRPTEN (interrupt enable) bit, [1-252](#)
 - IRPTL interrupt latch register, [1-271](#)
 - iso646.h header file, [1-25](#)
 - ISRs
 - called by interrupt dispatcher, [1-248](#)
 - in seg_rth memory section, [1-271](#)
 - receiving interrupt number, [1-248](#)
 - writing in C, [1-162](#)
 - I (start include directory) compiler switch, [1-37](#)
 - iteration interval, [2-65](#)
- ## K
- keywords
 - alternate, [1-45](#)
 - compiler, [1-105](#), [1-106](#)
 - extensions, not recognized, [1-45](#)
 - extensions, recognized, [1-105](#)
 - keywords (compiler), *See* compiler C/C++ extensions
- ## L
- _LANGUAGE_C macro, [1-262](#)
 - language extensions (compiler), *See* compiler C/C++ extensions
 - LDF, migrating from previous VisualDSP++ versions, [1-254](#)
 - LDF (linker description file) symbols, [1-277](#)
 - for managing stack and heap, [1-279](#)
 - ldf_heap_end, [1-279](#)
 - ldf_heap_length, [1-279](#)
 - ldf_heap_space, [1-279](#)
 - ldf_stack_end, [1-279](#)
 - ldf_stack_length, [1-279](#)
 - ldf_stack_space, [1-279](#)
 - leaf assembly routines, [1-329](#)
 - leaf_entry macro, [1-301](#), [1-317](#)
 - leaf_exit macro, [1-302](#), [1-317](#)

- legacy code, [1-137](#)
- library
 - building with elfar, [1-27](#)
 - optimization, [1-84](#)
 - searching for functions and global variables when linking, [1-39](#)
- library file, producing with elfar, [1-27](#)
- __lib_setup_processor routine, [1-275](#)
- lightweight debugging information, [1-36](#)
- line breaks, in string literals, [1-224](#)
- line debugging, [1-35](#)
- __LINE__ macro, [1-262](#)
- linker description file (.ldf file), [1-64](#)
- linking pragmas, [1-195](#)
- link library, [1-39](#)
- list-workarounds (supported errata workarounds) compiler switch, [1-40](#)
- live register, [2-56](#)
- L (library search directory) compiler switch, [1-39](#)
- l (link library) compiler switch, [1-39](#)
- long
 - identifier, [1-141](#)
 - latencies, avoiding, [2-38](#)
 - storage format, [1-307](#)
- long file names, handling with the
 - write-files switch, [1-69](#)
- _LONG keyword, [1-159](#)
- loop
 - annotations, [2-98](#)
 - avoiding array writes, [2-33](#)
 - avoiding conditional code in, [2-34](#)
 - avoiding function calls in, [2-35](#)
 - avoiding non-unit strides, [2-35](#)
 - control variables, [2-36](#)
 - cycle count, [2-88](#)
 - epilog, [2-57](#)
 - exit test, [2-36](#)
 - flattening, [2-95](#)
 - identification, [2-87](#)
 - identification annotation, [2-88](#)
 - inner vs. outer, [2-33](#)
 - invariant, [2-57](#)
 - iteration count, [2-50](#)
 - kernel, [2-56](#)
 - optimization, concepts, [2-58](#)
 - optimization, explained, [2-55](#)
 - optimization, pragmas, [1-166](#), [2-50](#)
 - optimization, terminology, [2-55](#)
 - parallel processing, [1-171](#)
 - prolog, [2-57](#)
 - register usage, [2-89](#)
 - resource usage, [2-89](#)
 - rotation, defined, [2-59](#)
 - rotation by hand, [2-32](#)
 - short, [2-30](#)
 - trip count, [2-35](#), [2-94](#)
 - unrolling, [2-30](#)
 - vectorization, [1-166](#), [2-51](#), [2-62](#)
- loop annotation information
 - disabling, [1-43](#)
 - enabling, [1-26](#)
- loop-carried dependency, [2-31](#), [2-32](#)
 - avoiding, [2-31](#)
- loop optimization pragmas, [2-50](#)
- L registers, [1-179](#), [1-294](#)
- L (search library) compiler switch, [1-48](#)

INDEX

lvalue

- GCC generalized, [1-222](#)
- generalized, [1-222](#)

M

macros

- ccall, [1-330](#)
- compound statements as, [1-265](#)
- defining, [1-28](#)
- expanding to a compound statement, [1-266](#)
- __HOSTNAME__, [1-64](#)
- interface support, [1-321](#)
- mixed C/C++ assembly support, [1-316](#)
- predefined preprocessor, [1-259](#)
- __RTTI, [1-76](#)
- SKIP_SPACES, [1-265](#)
- stack management, [1-330](#)
- __SYSTEM__, [1-64](#)
- __USERNAME__, [1-64](#)
- variable argument, [1-224](#)
- writing, [1-264](#)
- make rules only, [1-40](#)
- malloc (allocate uninitialized memory)
 - function, [1-283](#)
- map files, .XML files, [1-41](#)
- map (generate a memory map) compiler switch, [1-41](#)
- maximum performance, [2-44](#)
- MD (make and compile) compiler switch, [1-40](#)

mem21k initializer

- disabling, [1-45](#)
- not invoking after linking, [1-45](#)
- processing executable file, [1-274](#), [1-275](#)
- processing PROGBITS sections, [1-275](#)
- processing seg_init initialization section, [1-274](#)
- processing ZERO_INIT sections, [1-275](#)
- running, [1-41](#)
- MEM_ARGV memory section, [1-283](#)
- mem (enable memory initialization)
 - compiler switch, [1-41](#)
- memmove (move memory range) function, [1-63](#)

memory

- allocation, for stacks and heaps, [1-277](#)
- bank pragmas, [1-210](#)
- data placement in, [2-26](#)
- initialization, [1-41](#), [1-274](#)
- keywords, [1-132](#)
- map file, [1-41](#)
- maximum performance, [2-26](#)
- placing code in, [1-270](#)
- section names, [1-270](#)
- space assignments, [1-132](#)
- used for placing code in, [1-270](#)
- memory initialization
 - disabling, [1-45](#)
 - enabling, [1-41](#)
- memory initializer, *See* mem21k initializer
- memory keywords
 - function arguments and, [1-133](#)
 - function declarations with pointers, [1-133](#)
 - macros and, [1-134](#)
- memory map, generating, [1-41](#)
- memory-mapped registers (MMR),
 - accessing using macros, [1-129](#)
- minimum code size, compiling for, [2-43](#)

MISRA C

- rule 10.5 (required), [1-97](#)
- rule 12.12 (required), [1-98](#)
- rule 12.4 (required), [1-98](#)
- rule 12.8 (required), [1-98](#)
- rule 13.2 (advisory), [1-98](#)
- rule 13.7 (required), [1-99](#)
- rule 1.5 (required), [1-95](#)
- rule 16.10 (required), [1-99](#)
- rule 16.2 (required), [1-99](#)
- rule 16.4 (required), [1-99](#)
- rule 17.1 (required), [1-100](#)
- rule 17.2 (required), [1-100](#)
- rule 17.3 (required), [1-100](#)
- rule 17.4 (required), [1-100](#)
- rule 17.6 (required), [1-100](#)
- rule 18.2 (required), [1-101](#)
- rule 19.15 (advisory), [1-101](#)
- rule 19.7 (advisory), [1-101](#)
- rule 20.10 (required), [1-102](#), [1-103](#)
- rule 20.11 (required), [1-103](#)
- rule 20.3 (required), [1-101](#)
- rule 20.4 (required), [1-102](#)
- rule 20.7 (required), [1-102](#)
- rule 20.8 (required), [1-102](#)
- rule 20.9 (required), [1-102](#)
- rule 21.1 (required), [1-103](#)
- rule 2.4 (advisory), [1-95](#)
- rule 5.1 (required), [1-96](#)
- rule 5.5 (advisory), [1-96](#)
- rule 5.7 (advisory), [1-96](#)
- rule 6.3 (advisory), [1-96](#)
- rule 6.4 (advisory), [1-96](#)
- rule 8.10 (required), [1-97](#)
- rule 8.1 (required), [1-96](#)
- rule 8.5 (required), [1-96](#)
- rule 8.8 (required), [1-97](#)
- rule 9.1 (required), [1-97](#)
- rule clarifications, [1-92](#)

MISRA-C

- compiler, [1-91](#)
- compliance, [1-92](#)
 - rule 1.4 (required), [1-95](#)
 - rules, [1-95](#)
- MISRA C compiler, [1-91](#)
- misra C compiler switch, [1-71](#)
- MISRA-C switches, [1-71](#)
 - .misra extension files, [1-96](#)
 - .misra files, [1-71](#), [1-97](#)
 - misra-linkdir C compiler switch, [1-71](#)
 - misra-no-cross-module C compiler switch, [1-71](#)
 - misra-no-runtime C compiler switch, [1-71](#)
- MISRARepository directory, [1-71](#)
- _MISRA_RULES macro, [1-262](#)
- misra-strict C compiler switch, [1-72](#)
- misra-suppress-advisory C compiler switch, [1-72](#)
- misra-suppress-testing C compiler switch, [1-72](#)
- misra_types.h header file, [1-98](#)
- missing operands, in conditional expressions, [1-223](#)
- mixed C/C++ assembly naming conventions, [1-321](#)
- mixed C/C++ assembly programming
 - arguments and return, [1-304](#)
 - asm() constructs, [1-113](#), [1-115](#), [1-118](#), [1-124](#)
 - call preserved registers, [1-295](#)
 - compiler registers, [1-293](#)
 - data storage and type sizes, [1-307](#)
 - examples, [1-328](#)
 - return address, [1-330](#)
 - scratch registers, [1-296](#)
 - stack registers, [1-297](#)
 - stack usage, [1-298](#)
 - user registers, [1-294](#)

INDEX

- mixed C/C++ assembly programming
 - calling assembler subroutines, [1-311](#)
- mixed C/C++ assembly support, macros, [1-316](#)
- M (make only) compiler switch, [1-40](#)
- MMASK register, [1-293](#)
- MM (make rules and compile) compiler switch, [1-41](#)
- MODE1 register, [1-164](#), [1-165](#), [1-296](#)
- modulo
 - variable expansion unroll factor, [2-64](#)
- modulo scheduled loop, [2-98](#)
- modulo scheduling, [2-64](#), [2-65](#)
 - producing scheduled loops with, [2-64](#), [2-98](#)
- modulo scheduling information, [2-98](#)
- modulo variable expansion factor, [2-74](#)
- modulus array references, [1-151](#)
- Mo (processor output file) compiler switch, [1-41](#)
- Mt filename (output make rule) compiler switch, [1-41](#)
- multicore support, [1-195](#)
- multi-line asm() C program constructs, [1-124](#)
- multiline compiler switch, [1-41](#)
- multiple
 - attributes, [1-33](#)
 - heaps, [1-283](#), [1-291](#)
 - lines, spanning, [1-41](#)
- multi-statement macros, [1-265](#)
- N**
- namespace std, [1-75](#)
- naming conventions
 - assembly and C, [1-321](#)
 - assembly and C/C ++, [1-321](#)
- nested hardware loops, restrictions, [1-59](#)
- never-inline compiler switch, [1-42](#)
- newline, in string literals, [1-41](#), [1-46](#)
- new operator, with multiple heaps, [1-291](#)
- no-aligned-stack (do not align stack) compiler switch, [1-42](#)
- no-alttok (disable tokens) C++ mode compiler switch, [1-42](#)
- no-anach (disable C++ anachronisms) compiler switch, [1-76](#)
- no-annotate (disable assembly annotations) compiler common switch, [1-42](#)
- no-annotate-loop-instr compiler common switch, [1-43](#)
- no-auto-attrs compiler switch, [1-43](#)
- __NO_BUILTIN macro, [1-43](#), [1-263](#)
- no-builtin (no built-in functions) compiler switch, [1-43](#)
- no-builtin (no built-in functions switch, [1-43](#)
- no-circbuf (no circular buffer) compiler switch, [1-44](#)
- no-const-strings compiler switch, [1-44](#)
- no-db (no delayed branches) compiler switch, [1-44](#)
- no-def (disable definitions) compiler switch, [1-44](#)
- no-eh (disable exception handling) C++ mode compiler switch, [1-45](#)
- no-extra-keywords (not quite -ansi) compiler switch, [1-45](#)
- no-fp-associative compiler switch, [1-45](#)
- no implicit inclusion, defined, [1-190](#)
- no-implicit-inclusion C++ mode compiler switch, [1-76](#)
- NO_INIT qualifier, [1-204](#)
- __NO_LONGLONG macro, [1-263](#)
- no-mem (disable memory initialization) compiler switch, [1-45](#)
- no-multiline compiler switch, [1-46](#)
- non-constant initializer support (compiler), [1-142](#)

non-default heap, [1-286](#)
 non-leaf
 assembly routines, [1-329](#)
 routines to make calls (RMS), [1-336](#)
 non-temporary files location, [1-54](#)
 non-unit strides, avoiding in loops, [2-35](#)
 -no-progress-rep-timeout compiler switch, [1-46](#)
 normal interrupt dispatcher
 described, [1-249](#)
 saving data, index, modify, base registers, [1-249](#)
 -normal-word-code compiler switch, [1-49](#)
 __NORMAL_WORD_CODE__ macro, [1-263](#)
 -no-rtti (disable run-time type identification) C++ mode compiler switch, [1-76](#)
 -no-sat-associative compiler switch, [1-46](#)
 -no-saturation (no faster operations) compiler switch, [1-46](#)
 -no-shift-to-add compiler switch, [1-47](#)
 -no-simd (disable SIMD mode) compiler switch, [1-47](#)
 -no-std-ass (disable standard assertions) compiler switch, [1-47](#)
 -no-std-def (disable standard definitions) compiler switch, [1-48](#)
 -no-std-inc (disable standard include search) compiler switch, [1-48](#)
 -no-std-lib (disable standard library search) compiler switch, [1-48](#)
 -no-std-templates C++ mode compiler switch, [1-76](#)
 -no-threads (disable thread-safe build) compiler switch, [1-48](#)
 -no-workaround (workaround id) compiler switch, [1-48](#)
 num variable, [1-51](#)
 -nwc compiler switch, [1-49](#)

O

-Oa (automatic function inlining) compiler switch, [1-49](#)
 \$OBS_LIBS_INTERNAL macro, [1-354](#)
 -O (enable optimization) compiler switch, [1-49](#)
 -Og (optimize while preserving debugging information) compiler switch, [1-50](#)
 -o (output) compiler switch, [1-52](#)
 operand constraints, [1-119](#)
 operation extensions, [1-107](#)
 optimization
 code size, [1-50](#), [2-43](#)
 compiler, [2-4](#)
 configurations (or levels), [1-80](#)
 controlling, [1-80](#)
 debug information generation enabled, [2-8](#)
 default, [1-81](#)
 disabling, [1-49](#)
 enabling, [1-49](#), [1-84](#)
 for code size, [2-43](#)
 for maximum performance, [2-44](#)
 inlining process and, [1-111](#)
 inner loop, [2-33](#)
 interprocedural analysis (IPA), [1-84](#)
 library, [1-84](#)
 loops, [1-166](#)
 pragmas used in, [2-45](#)
 preserving debugging information, [1-50](#)
 reporting progress in, [1-58](#)
 sliding scale for, [1-51](#)
 speed, [1-50](#), [2-43](#)
 speed versus size, [1-50](#)
 switches, [1-49](#), [2-54](#)
 with interprocedural analysis (IPA), [1-84](#)
 optimization and debugging, enabling, [1-50](#)

INDEX

- optimization levels
 - automatic inlining, [1-82](#)
 - debug, [1-81](#)
 - default, [1-81](#)
 - interprocedural optimizations, [1-83](#)
 - PGO, [1-82](#)
 - procedural optimizations, [1-81](#)
- Os (optimize for size) compiler switch, [1-50](#)
- outer loops, [2-33](#)
- out-of-line copy, [1-111](#)
- output operand, of asm() construct, [1-116](#), [1-126](#)
- overlay-clobbers compiler switch, [1-53](#)
- overlay pragma, [1-183](#)
- overlay (program may use overlays)
 - compiler switch, [1-52](#)
- overlays, registers clobbered by overlay manager, [1-53](#)
- overlays, using in program, [1-52](#)
- Ov num (optimize for speed versus size)
 - compiler switch, [1-50](#)

P

- passing
 - arguments to driver, [1-62](#)
 - function parameters, [1-304](#)
- path-install (installation location)
 - compiler switch, [1-54](#)
- path-output (non-temporary files location) compiler switch, [1-54](#)
- path-temp (temporary files location)
 - compiler switch, [1-54](#)

- path- (tool location) compiler switch, [1-54](#)
- pchdir (locate PCHRepository) compiler switch, [1-55](#)
- pch (precompiled header) compiler switch, [1-55](#)
- PCHRepository directory, [1-55](#)
- peeled iterations, [2-95](#)
- per-file optimizations, [1-81](#), [1-83](#)
- performance optimization, [1-50](#)
- .pgi file, [2-12](#)
- PGO
 - See also* profile-guided optimization (PGO)
 - collecting data, [1-82](#)
 - data sets, [2-12](#)
 - data sets, multiple, [2-12](#)
 - operation via menu selection, [1-82](#)
 - session identifier, [1-55](#)
 - supported in the simulator only, [1-82](#), [2-9](#)
- .pgo file, [1-82](#)
 - from wrapper project, [2-11](#)
 - gathering data with the -pguide switch, [1-56](#)
 - in PGO process, [1-55](#), [1-82](#), [2-10](#)
 - session-id identifier, [1-55](#)
- pgo-session session-id compiler switch, [1-55](#)
 - used to separate profiles, [2-11](#)
- pguide (profile-guided optimization)
 - compiler switch, [1-56](#)
- pipeline viewer, [2-38](#)

- placement
 - all data, [1-61](#), [1-138](#)
 - constant data, [1-61](#), [1-138](#)
 - constant data declared with `_pm` keyword, [1-61](#), [1-138](#)
 - C++ virtual lookup table, [1-61](#), [1-138](#)
 - data, [1-60](#), [1-137](#)
 - initialized data declared with `_pm` keyword, [1-61](#), [1-138](#)
 - initialized variable data, [1-61](#), [1-138](#)
 - initializing aggregate autos, [1-61](#), [1-138](#)
 - jump-tables used to implement C/C++ switch statements, [1-61](#), [1-138](#)
 - machine instructions, [1-61](#), [1-138](#)
 - of run-time library functions, [1-348](#)
 - static C++ class constructor functions, [1-61](#), [1-138](#)
 - string literals, [1-61](#), [1-138](#)
 - zero-initialized variable data, [1-61](#), [1-138](#)
- placement support keyword (section), [1-136](#)
- `pm`, *See* dual memory support keywords (`pm`, `dm`)
- `pm_constdata` section identifier, [1-61](#), [1-138](#)
- PMDA access, [1-88](#)
- `pm_data` section identifier, [1-61](#), [1-138](#)
- PM qualifier, [1-204](#)
- pointer
 - aligned on dual-word boundaries, [2-20](#)
 - and index styles, [2-23](#)
 - arithmetic action on, [1-225](#)
 - class support keyword (`restrict`), [1-106](#), [1-139](#)
 - incrementing, [2-23](#)
 - induction variable, [1-167](#)
 - resolving aliasing, [2-37](#)
 - to data that is aligned, [2-19](#)
- pointer-induction variables, [1-167](#)
- pointer registers, [1-297](#)
- P (omit line numbers and compile) compiler switch, [1-53](#), [1-54](#)
- POP STS instruction, [1-163](#), [1-164](#)
- pplist (preprocessor listing) compiler switch, [1-56](#)
- `#pragma alignment_region`, [1-160](#)
- `#pragma alignment_region_end`, [1-160](#)
- `#pragma align_num`, [1-158](#), [1-167](#), [2-19](#)
- `#pragma all_aligned`, [2-52](#)
- `#pragma alloc`, [1-173](#), [2-45](#)
- `#pragma always_inline`, [1-26](#), [1-109](#), [1-192](#)
- `#pragma avoid_anomaly_45`, [1-217](#)
- `#pragma bank_memory_kind`, [1-214](#)
- `#pragma bank_optimal_width`, [1-216](#)
- `#pragma bank_read_cycles`, [1-215](#)
- `#pragma bank_write_cycles`, [1-215](#)
- `#pragma can_instantiate_instance`, [1-189](#)
- `#pragma code_bank`, [1-211](#)
- `#pragma compiler_support`, [1-166](#)
- `#pragma const`, [1-174](#), [2-46](#)
- `#pragma core`, [1-195](#)
- `#pragma data_bank`, [1-212](#)
- `#pragma default_section`, [1-201](#), [1-246](#), [1-282](#)
- `#pragma diag`, [1-207](#), [2-7](#)
- `#pragma diag(errors)`, [1-209](#)
- `#pragma diag(pop)`, [1-209](#)
- `#pragma diag(push)`, [1-209](#)
- `#pragma diag(remarks)`, [1-209](#)
- `#pragma diag(warnings)`, [1-209](#)
- `#pragma do_not_instantiate_instance`, [1-189](#)
- `#pragma file_attr`, [1-206](#)
- `#pragma generate_exceptions_tables`, [1-218](#)
- `#pragma hdrstop`, [1-189](#)
- `#pragma inline`, [1-109](#), [1-110](#), [1-194](#)
- `#pragma instantiate`, [1-344](#)
- `#pragma instantiate_instance`, [1-188](#)
- `#pragma interrupt`, [1-163](#)

INDEX

#pragma interrupt_complete, [1-165](#)
#pragma interrupt_complete_nesting,
 [1-164](#)
pragma interrupt dispatcher
 described, [1-250](#)
 zeroing Length registers, [1-251](#)
#pragma linkage_name, [1-195](#)
#pragma loop_count(min, max, modulo),
 [1-167, 2-50](#)
#pragma loop_unroll N, [1-168](#)
#pragma misra_func, [1-174](#)
#pragma no_alias, [1-170, 2-53](#)
#pragma no_implicit_inclusion, [1-190](#)
#pragma no_pch, [1-191](#)
#pragma noreturn, [1-174](#)
#pragma no_vectorization, [1-167, 2-51](#)

#pragma once, [1-192](#)
#pragma optimize_as_cmd_line, [1-172](#)
#pragma optimize_for_space, [1-172,](#)
 [1-194, 2-49](#)
#pragma optimize_for_speed, [1-172, 2-49](#)
#pragma optimize_off, [1-172, 2-49](#)
#pragma overlay, [1-183](#)
#pragma pack (alignopt), [1-161](#)
#pragma pad (alignopt), [1-162](#)
#pragma param_never_null, [1-184](#)
#pragma pgo_ignore, [1-175](#)
#pragma pure, [1-175, 2-46](#)
#pragma regs_clobbered, [1-176, 2-47](#)
#pragma regs_clobbered_call, [1-180](#)
#pragma result_alignment, [1-184, 2-47](#)
#pragma retain_name, [1-200](#)

- pragmas
 - alignment_region, 1-160
 - alignment_region_end, 1-160
 - align_num, 1-158, 1-167
 - alloc, 1-173
 - always_inline, 1-192
 - avoid_anomaly_45, 1-217
 - bank_memory_kind, 1-214
 - bank_optimal_width, 1-216
 - bank_read_cycles, 1-215
 - bank_write_cycles, 1-215
 - can_instantiate instance, 1-189
 - code_bank, 1-211
 - const, 1-174
 - core, 1-195
 - data alignment, 1-158
 - data_bank, 1-212
 - default_section, 1-201, 1-282
 - diag, 1-207
 - do_not_instantiate instance, 1-189
 - file_attr, 1-206
 - function side-effect, 1-173
 - generate_exceptions_tables, 1-218
 - hdrstop, 1-189
 - header file control, 1-189
 - inline, 1-109, 1-110, 1-194
 - instantiate instance, 1-188
 - interrupt, 1-163
 - interrupt_complete, 1-165
 - interrupt_complete_nesting, 1-164
 - interrupt handler, 1-162
 - interrupt vector table, 1-165
 - linkage_name, 1-195
 - linking, 1-195
 - linking control, 1-195
 - loop_count (min, max, modulo), 1-167
 - loop optimization, 1-166, 2-50
 - loop_unroll N, 1-168
 - memory bank, 1-210
 - misra_func, 1-174
 - never_inline, 1-194
 - no_alias, 1-170
 - no_implicit_inclusion, 1-190
 - no_pch, 1-191
 - noreturn, 1-174
 - no_vectorization, 1-167
 - once, 1-192
 - optimize_as_cmd_line, 1-172, 1-209
 - optimize_for_space, 1-172, 1-209
 - optimize_off, 1-172
 - overlay, 1-183
 - pack (alignopt), 1-161
 - pad (alignopt), 1-162
 - param_never_null, 1-184
 - pgo_ignore, 1-175
 - pure, 1-175
 - regs_clobbered_call, 1-180
 - regs_clobbered_string, 1-176
 - result_alignment, 1-184
 - retain_name, 1-200
 - section, 1-201, 1-282
 - SIMD_for, 1-166, 1-237
 - stack_bank, 1-213
 - suppress_null_check, 1-186
 - syntax, 1-156
 - system_header, 1-192
 - template instantiation, 1-187
 - used for code optimization, 2-45
 - vector_for, 1-171
 - weak_entry, 1-206
- #pragma section, 1-137, 1-201, 1-246, 1-282
- #pragma SIMD_for, 1-166, 1-237, 2-52
- #pragma stack_bank, 1-213
- #pragma suppress_null_check, 1-186
- #pragma system_header, 1-192
- #pragma vector_for, 1-171, 2-51
- #pragma weak_entry, 1-206
- precompiled header files, generating and use, 1-55

INDEX

- precompiled header repository, locating, [1-55](#)
- predefined macros
 - _MISRA_RULES, [1-71](#), [1-262](#)
- predefined preprocessor macros, [1-259](#)
- prefersMem attribute, [1-351](#)
- prefersMemNum attribute, [1-351](#)
- prelinker, [1-84](#), [1-345](#), [2-12](#)
 - MISRA-C compiler, [1-97](#)
- preprocessor
 - commands, [1-258](#)
 - listing file, [1-56](#)
 - predefined macros, [1-259](#)
 - program, [1-258](#)
 - warnings, [1-145](#)
- primary register set, [1-296](#)
- procedural optimizations, [1-81](#)
- procedure statistics, [2-82](#)
- processor selection, [1-57](#)
- proc processor (target processor) compiler switch, [1-57](#)
- profile-guided optimization
 - #pragma pgo_ignore, [1-175](#)
- profile-guided optimization (PGO)
 - adding instrumentation, [1-56](#)
 - command-line arguments in, [1-283](#)
 - common scenario, [1-82](#)
 - described briefly, [1-82](#)
 - multiple PGO data sets, [2-12](#)
 - multiple source uses, [2-11](#)
 - non-simulatable applications, [2-10](#)
 - operation via menu selection, [1-82](#)
 - Ov num switch, [1-52](#), [2-12](#), [2-44](#)
 - PGO session identifier, [1-55](#)
 - pgo-session id switch, [1-55](#)
 - run-time behavior, [2-8](#)
 - simulator, [2-9](#)
 - when not used, [1-52](#)
 - when to use, [2-8](#), [2-13](#)

- profile instrumentation, and profile-guided optimization (PGO), [1-56](#)
- PROGBITS section, [1-275](#)
- program
 - assignments, [1-132](#)
 - memory code storage, [1-271](#)
 - memory data storage, [1-272](#)
- progress-rep-func compiler switch, [1-57](#)
- progress-rep-opt compiler switch, [1-58](#)
- progress reporting, [1-58](#)
- progress-rep-timeout compiler switch, [1-58](#)
- progress-rep-timeout-secs compiler switch, [1-58](#)
- prototype, incomplete, [1-68](#)
- PUSH STS instruction, [1-163](#), [1-164](#), [1-252](#)
- puts macro, [1-318](#)

Q

- _QUAD keyword, [1-159](#)
- QUALIFIER keywords, for section pragma, [1-204](#)

R

- RAM, initializing, [1-275](#)
- R- (disable source path) compiler switch, [1-59](#)
- read_extmem function, [1-247](#)
- reads macro, [1-318](#)
- realloc heap function, [1-283](#)
- reductions, [2-31](#)
- ref-code characters, [1-70](#)
- register information, disabling propagation of, [1-52](#), [1-183](#)

- registers
 - alternate, [1-297](#)
 - asm() constructs, [1-118](#)
 - assigning to operands, [1-119](#)
 - call preserved, [1-295](#)
 - clobbered by overlay manager, [1-53](#)
 - clobbered register sets table, [1-176](#), [1-177](#)
 - compiler, [1-293](#)
 - live, [2-56](#)
 - performance, [1-294](#)
 - pointer, [1-297](#)
 - reserved, [1-59](#)
 - reserving, [1-294](#)
 - return, [1-180](#)
 - scratch, [1-296](#)
 - soft-wired, [1-179](#)
 - stack, [1-297](#)
 - transfer, [1-304](#)
 - unclobbered, [1-178](#)
 - user, [1-294](#)
 - user-reserved, [1-179](#)
 - registers for arguments and return (add 2)
 - example, [1-335](#)
 - register usage, *See* mixed C/C++ assembly programming
 - regs_clobbered string, [1-177](#)
 - remark keyword, [1-208](#)
 - remarks
 - control pragma, [1-207](#)
 - using in diagnostics, [2-6](#)
 - RESERVE_EXPAND() LDF command, [1-278](#)
 - RESERVE() LDF command, [1-278](#)
 - reserve (reserve register) compiler switch, [1-59](#), [1-294](#)
 - reset_saturate_mode function, [1-231](#)
 - RESOLVE LDF directive, [1-236](#)
 - restore keyword, [1-208](#)
 - restore_reg macro, [1-319](#)
 - restrict
 - See also* pointer class support keyword
 - keyword, [2-37](#)
 - operator keyword, [1-139](#)
 - qualifier, [2-37](#)
 - restricted pointer, [2-37](#)
 - restrict-hardware-loops compiler switch, [1-59](#)
 - return registers, [1-180](#)
 - return value transfer, [1-304](#)
 - rframe instruction, [1-89](#), [1-301](#)
 - R (search for source files) compiler switch, [1-58](#)
 - rtti (enable run-time type identification)
 - C++ mode compiler switch, [1-76](#)
 - __RTTI macro, [1-76](#), [1-263](#)
 - run-time
 - C/C++ environment, *See* mixed C/C++ assembly programming
 - C header, [1-275](#)
 - checking, [1-103](#)
 - default header file, [1-275](#)
 - disabling type identification, [1-76](#)
 - dynamically allocate/deallocate memory, [1-273](#)
 - enabling type identification, [1-76](#)
 - header, default, [1-310](#)
 - header, source code for, [1-310](#)
 - header storage, [1-275](#)
 - heap memory, [1-273](#)
 - stack, [1-297](#)
 - stack memory, [1-273](#)
 - RUNTIME_INIT qualifier, [1-204](#)
- S
- sat-associative compiler switch, [1-60](#)
 - saturated arithmetic, [1-230](#)

INDEX

- saturation
 - disabling, [1-46](#)
 - disabling associativity, [1-46](#)
 - enabling associativity, [1-60](#)
- save_reg macro, [1-318](#)
- save-temps compiler switch, [1-84](#)
- save-temps (save intermediate files)
 - compiler switch, [1-60](#)
- scalar variables, [2-31](#)
- scheduling, of program instructions, [2-56](#)
- S compiler switch, [1-84](#)
- scratch registers, [1-296](#)
- scratch registers (dot oroduct) example,
[1-332](#)
- search path
 - for include files, [1-36](#)
 - for library files, [1-39](#)
 - for library files when linking, [1-39](#)
- secondary registers, [1-293](#), [1-297](#)
- section
 - elimination, [2-43](#)
 - keyword, [1-106](#), [1-136](#), [1-270](#)
 - names, [1-270](#)
 - placing symbols in, [1-201](#)
 - qualifiers, [1-201](#)
- .SECTION assembler directive, [1-136](#),
[1-270](#)
- section id (data placement) compiler
switch, [1-60](#), [1-137](#), [1-282](#)
- section identifiers, compiler-controlled,
[1-61](#), [1-137](#)
- section pragmas, [1-201](#)
- SECTKIND keywords, [1-202](#)
- SECTSTRING double-quoted string,
[1-203](#)
- seg_dmda data section, [1-272](#), [1-273](#)
- seg_dmda memory section, [1-270](#)
- seg_heap declaration, [1-274](#)
- seg_heap heap, [1-284](#)
- seg_heap memory section, [1-271](#)
- seg_heap section, [1-273](#)
- seg_heaq heap, [1-284](#)
- seg_init.asm file, [1-284](#)
- seg_init.doj file, [1-284](#)
- seg_init initialization section, [1-274](#)
- seg_init memory section, [1-271](#)
- seg_int_code memory section, [1-271](#)
- seg_int_code_sw memory section, [1-271](#)
- segment
 - See also* placement support keyword
(section)
 - keyword, *See* section keyword
 - legacy keyword, [1-137](#)
- seg_pmco code section, [1-271](#)
- seg_pmco memory section, [1-270](#)
- seg_pmda data section, [1-272](#)
- seg_pmda memory section, [1-271](#)
- seg_rth memory section, [1-271](#)
- seg_rth run-time header section, [1-275](#)
- seg_stak memory section, [1-271](#)
- seg_stak section, [1-273](#)
- seg_swco memory section, [1-270](#)
- self-modifying code, avoiding, [1-251](#)
- set_alloc_type function, [1-289](#)
- set_saturate_mode function, [1-231](#)
- shadow register operation, [1-239](#)
- shift-to-add conversion, disabling, [1-47](#)
- short-form keywords
 - disabling, [1-45](#)
 - enabling, [1-33](#)
- short storage format, [1-307](#)
- short-word-code compiler switch, [1-62](#)
- __SHORT_WORD_CODE__ macro,
[1-263](#)
- show (display command line) compiler
switch, [1-62](#)
- signalcb() function, [1-248](#)
- signalf() function, [1-249](#)
- signalfnsn() function, [1-251](#)
- signal() function, [1-249](#)

- signals() function, [1-250](#)
- signalss() function, [1-250](#)
- signed-bitfield (make plain bit-fields signed) compiler switch, [1-62](#)
- __SIGNED_CHARS__ macro, [1-263](#)
- silicon revision
 - management, [1-85](#)
 - version setting, [1-86](#)
- silicon revision, specifying, [1-62](#), [1-86](#)
- __SILICON_REVISION__ macro, [1-87](#)
- SIMD_for loop, restrictions, [1-239](#)
- SIMD_for pragma, [1-166](#), [1-233](#), [1-237](#), [1-242](#)
- SIMD mode
 - 32-bit external bus use, [1-236](#)
 - anomaly #40, [1-239](#)
 - automatically generating SIMD code, [1-232](#)
 - avoiding anomaly #40, [1-240](#)
 - C/C++ callable subroutines, [1-324](#)
 - C/C++ constraints in, [1-238](#)
 - C/C++ data alignment rules, [1-245](#)
 - data alignment, [1-235](#), [1-244](#)
 - data increments, [1-242](#)
 - defined, [1-231](#)
 - disabling, [1-47](#)
 - disabling automatic SIMD generation, [1-233](#)
 - double-word boundary alignment, [1-235](#)
 - fetches, [1-235](#)
 - loop counter rules, [1-244](#)
 - misalignment, [1-236](#)
 - odd locations, [1-236](#)
 - performance in C/C++, [1-240](#)
 - processing multichannel data in, [1-233](#)
 - processing single channel data in, [1-234](#)
 - restrictions, [1-235](#)
 - SIMD_for pragma, [1-237](#)
 - subroutines, [1-325](#)
 - usage examples, [1-242](#)
- __SIMDSHARC__ macro, [1-260](#), [1-261](#), [1-263](#)
- simulator, used with PGO, [1-82](#), [2-9](#)
- single case range, [1-225](#)
- sinking process, [2-58](#)
- si-revision (silicon revision) compiler switch, [1-62](#), [1-86](#)
- SISD mode, [1-232](#), [1-234](#)
- sizeof() operator, [1-141](#), [1-225](#)
- SKIP_SPACES macro, [1-265](#)
- sliding scale, between 0 and 100, [1-51](#)
- slow floating-point emulation software, [1-30](#)
- small applications, producing, [2-43](#)
- software pipelining, [2-59](#), [2-62](#)
- source annotations, [2-7](#)
- source code, checking for syntax errors, [1-64](#)
- source directory, adding, [1-58](#)
- source path, disabling, [1-59](#)
- spill, to the stack, [2-56](#)
- S (stop after compilation) compiler switch, [1-60](#)
- s (strip debug information) compiler switch, [1-60](#)
- stack
 - frame, ADSP-21020 processor, [1-299](#)
 - frame,
 - ADSP-2106x/2116x/2126x/2136x processors, [1-301](#)
 - frame, defined, [1-298](#)
 - managing in memory, [1-298](#)
 - managing routines, [1-298](#)
 - managing with macros, [1-330](#)
 - memory allocation, [1-277](#)
 - pointer, [1-298](#)
 - registers, [1-297](#)
- stack alignment
 - disabling, [1-42](#)
 - enabling, [1-25](#)

INDEX

- stack for arguments and return (add 5)
 - example, [1-334](#)
- stage count (SC), [2-64](#), [2-69](#)
- standard
 - heap management functions, [1-283](#)
 - include search, disabling, [1-48](#)
 - interface, using with alternate heaps, [1-289](#)
 - library search, disabling, [1-48](#)
 - macro definitions, disabling, [1-48](#)
 - optimizations, [1-35](#)
- standard assertions
 - disabling, [1-47](#)
 - enabling, [1-23](#)
- standard functions
 - accessing alternate heaps by, [1-289](#)
 - list of, [1-283](#)
- statement expression, [1-220](#)
- static data, [1-272](#)
- statistical profiling, [2-8](#)
- status register, saving data in, [1-164](#)
- `__STDC__` macro, [1-263](#)
- `__STDC_VERSION__` macro, [1-263](#)
- std namespace, [1-75](#)
- `-std-templates` C++ mode compiler switch, [1-77](#)
- STI memory area, [1-282](#)
- sti section identifier, [1-61](#), [1-138](#)
- string, literals with line breaks, [1-224](#)
- string concatenation feature, [1-124](#)
- string literals
 - marking as const-qualified, [1-28](#)
 - multiline, [1-41](#)
 - no-multiline, [1-46](#)
 - not making const-qualified, [1-44](#)
- struct
 - assignment, [1-63](#)
 - copying, [1-63](#)
- `-structs-do-not-overlap` compiler switch, [1-63](#)
- struct/unions, [1-227](#)
- super-fast interrupt dispatcher
 - ADSP-2106x restriction, [1-252](#)
 - context switching, [1-298](#)
 - described, [1-250](#)
- suppress keyword, [1-208](#)
- `-swc` compiler switch, [1-63](#)

switches

- A (assert) compiler switch, [1-23](#)
- add-debug-libpaths, [1-24](#)
- aligned-stack (align stack), [1-25](#)
- alttok (alternative tokens), [1-25](#)
- always-inline, [1-26](#)
- annotate (enable assembly annotations), [1-26](#)
- annotate-loop-instr, [1-26](#)
- auto-attrs (automatic attributes), [1-27](#)
- build-lib (build library), [1-27](#)
- C (comments), [1-27](#)
- c (compile only), [1-27](#)
- compatible-pm-dm, [1-27](#)
- const-read-write, [1-28](#)
- const-strings, [1-28](#)
- D (define macro), [1-28](#)
- debug-types, [1-29](#)
- double-size-32|64, [1-29](#)
- double-size-any, [1-30](#)
- dryrun (terse dry-run), [1-31](#)
- dry (verbose dry-run), [1-31](#)
- ED (run after preprocessing to file), [1-31](#)
- EE (run after preprocessing), [1-31](#)
- enum-is-int, [1-33](#)
- E (stop after preprocessing), [1-31](#)
- extra-keywords (enable short-form keywords), [1-33](#)
- file-attr name, [1-33](#)
- @filename (command file), [1-23](#)
- flags (command-line input), [1-34](#)
- float-to-int, [1-34](#)
- force-circbuf, [1-34](#)
- fp-associative (floating-point associative operation), [1-35](#)
- full-version (display versions), [1-35](#)
- g (generate debug information), [1-35](#)
- glite (lightweight debugging), [1-36](#)
- help (command-line help), [1-36](#)
- HH (list headers and compile), [1-36](#)
- H (list headers), [1-36](#)
- I directory (include search directory), [1-36](#)
- i (less includes), [1-38](#)
- implicit-pointers, [1-38](#)
- include (include file), [1-39](#)
- ipa (interprocedural analysis), [1-39](#)
- I (start include directory), [1-37](#)
- list-workarounds (supported errata workarounds), [1-40](#)
- L (library search directory), [1-39](#)
- l (link library), [1-39](#)
- map filename (generate a memory map), [1-41](#)
- MD (generate make rules and compile), [1-40](#)
- mem (enable memory initialization), [1-41](#)
- M (generate make rules only), [1-40](#)
- MM (generate make rules and compile), [1-41](#)
- Mo (processor output file), [1-41](#)
- Mt filename (output make rule), [1-41](#)
- multiline, [1-41](#)
- never-inline, [1-42](#)
- no-aligned-stack (disable stack alignment), [1-42](#)
- no-alttok (disable alternative tokens), [1-42](#)
- no-annotate (disable alternative tokens), [1-42](#)
- no-annotate-loop-instr, [1-43](#)
- no-builtin (no built-in functions), [1-43](#)
- no-circbuf (no circular buffer), [1-44](#)
- no-const-strings, [1-44](#)
- no-db (no delayed branches), [1-44](#)
- no-defs (disable defaults), [1-44](#)
- no-extra-keywords (disable short-form keywords), [1-45](#)

INDEX

- no-fp-associative, [1-45](#)
- no-mem (disable memory initialization), [1-45](#)
- no-multiline, [1-46](#)
- no-progress-rep-timeout, [1-46](#)
- normal-word-code, [1-49](#)
- no-sat-associative, [1-46](#)
- no-saturation (no faster operations), [1-46](#)
- no-shift-to-add, [1-47](#)
- no-simd (disable SIMD mode), [1-47](#)
- no-std-ass (disable standard assertions), [1-47](#)
- no-std-def (disable standard macro definitions), [1-48](#)
- no-std-inc (disable standard include search), [1-48](#)
- no-std-lib (disable standard library search), [1-48](#)
- no-threads (disable thread-safe build), [1-48](#)
- no-workaround (workaround id), [1-48](#)
- nwc, [1-49](#)
- Oa (automatic function inlining), [1-49](#)
- O (enable optimizations), [1-49](#)
- Og (optimize while preserving debugging information), [1-50](#)
- o (output file), [1-52](#)
- Os (optimize for size), [1-50](#)
- overlay, [1-52](#)
- overlay-clobbers, [1-53](#)
- Ov num (optimize for speed vs. size), [1-50](#)
- path-install (installation location), [1-54](#)
- path-output (non-temporary files location), [1-54](#)
- path-temp (temporary files location), [1-54](#)
- path- (tool location), [1-54](#)
- pchdir (locate PCHRepository), [1-55](#)
- pch (precompiled header), [1-55](#)
- pgo-session session-id, [1-55](#)
- pguide (profile-guided optimization), [1-56](#)
- P (omit line numbers and compile), [1-54](#)
- pplist (preprocessor listing), [1-56](#)
- PP (omit line numbers and compile), [1-54](#)
- proc processor, [1-57](#)
- progress-rep-func, [1-57](#)
- progress-rep-opt, [1-58](#)
- progress-rep-timeout, [1-58](#)
- progress-rep-timeout-secs, [1-58](#)
- R (add source directory), [1-58](#)
- R- (disable source path), [1-59](#)
- reserve (reserve register), [1-59](#)
- restrict-hardware-loops, [1-59](#)
- S, [1-84](#)
- sat-associative, [1-60](#)
- save-temps, [1-84](#)
- save-temps (save intermediate files), [1-60](#)
- section id (data placement), [1-60](#)
- short-word-code, [1-62](#)
- show (display command line), [1-62](#)
- signed-bitfield (make plain bit-fields signed), [1-62](#)
- si-revision version (silicon revision), [1-62](#), [1-86](#)
- sourcefile (parameter), [1-23](#)
- S (stop after compilation), [1-60](#)
- s (strip debug information), [1-60](#)
- structs-do-not-overlap, [1-63](#)
- swc, [1-63](#)
- syntax-only (check syntax only), [1-64](#)
- syntax-only (system definitions), [1-64](#)
- T filename (.ldf file), [1-64](#)
- threads (enable thread-safe build), [1-64](#)
- time (tell time), [1-65](#)

- unsigned-bitfield (make plain bit-fields unsigned), [1-65](#)
 - U (undefine macro), [1-65](#)
 - verbose, [1-66](#)
 - version (display version), [1-66](#)
 - v (version and verbose), [1-66](#)
 - warn-protos (warn if incomplete prototype), [1-68](#)
 - w (disable all warnings), [1-68](#)
 - Werror-limit (maximum compiler errors), [1-67](#)
 - Werror-warnings (treat warnings as errors), [1-67](#)
 - W{...} number (override error message), [1-67](#)
 - workaround workaround_id, [1-69](#), [1-88](#)
 - Wremarks (enable diagnostic warnings), [1-68](#)
 - write-files (enable driver I/O redirection), [1-69](#)
 - write-opts (user options), [1-69](#)
 - Wterse (enable terse warnings), [1-68](#)
 - xref (cross-reference list), [1-70](#)
 - switch section identifier, [1-61](#), [1-138](#)
 - symbolic names for individual bits, [1-150](#)
 - symbols, placing in sections, [1-201](#)
 - syntax-only (check syntax only) compiler switch, [1-64](#)
 - sysdef (system definitions) compiler switch, [1-64](#)
 - sysreg_bit_clr function, [1-147](#)
 - sysreg_bit_clr_nop function, [1-147](#)
 - sysreg_bit_* interrogation and manipulation functions, [1-150](#)
 - sysreg_bit_set function, [1-147](#)
 - sysreg_bit_set_nop function, [1-148](#)
 - sysreg_bit_tgl function, [1-148](#)
 - sysreg_bit_tgl_nop function, [1-148](#)
 - sysreg_bit_tst_all function, [1-149](#)
 - sysreg_bit_tst function, [1-148](#)
 - sysreg.h header file, [1-146](#), [1-147](#), [2-39](#)
 - sysreg_read function, [1-147](#)
 - sysreg_write function, [1-147](#)
 - sysreg_write_nop function, [1-147](#)
 - system initialization code, [1-271](#)
 - __SYSTEM__ macro, [1-64](#)
 - system macros, defined, [1-64](#)
 - system registers
 - accessing, [1-129](#), [1-147](#)
 - handling, [2-39](#)
 - list of, [1-149](#)
 - symbolic names for individual bits in, [1-150](#)
- T**
- target processor, specifying, [1-57](#)
 - template
 - class, [1-344](#)
 - control pragma, [1-190](#)
 - function, [1-344](#)
 - instantiation pragmas, [1-187](#)
 - support in C++, [1-344](#)
 - un-instantiated, [1-346](#)
 - template, of asm() construct, [1-115](#)
 - template instantiation, [1-344](#)
 - temporary files location, [1-54](#)
 - T filename (linker description file)
 - compiler switch, [1-64](#)
 - thread-safe
 - code, [1-65](#)
 - libraries, using with VDK, [1-65](#)
 - thread-safe build
 - disabling, [1-48](#)
 - enabling, [1-65](#)
 - threads (enable thread-safe build) compiler switch, [1-64](#)
 - __TIME__ macro, [1-263](#)
 - time (tell time) compiler switch, [1-65](#)
 - transfer registers, [1-304](#)

INDEX

transferring
 function arguments and return value, [1-304](#)
 function parameters to assembly routines, [1-304](#)
trip
 count, [2-64](#)
 loop count, [2-94](#)
 maximum, [2-65](#)
 minimum, [2-65](#)
 modulo, [2-64](#)
trip count, [2-76](#)
true, *See* Boolean type support keywords
 (bool, true, false)
type cast, [1-225](#)
typeof keyword, [1-221](#)
type sizes, data, [1-307](#)

U

unclobbered registers, [1-178](#)
unnamed struct/union fields, [1-227](#)
-unsigned-bitfield (make plain bit-fields unsigned) compiler switch, [1-65](#)
__USERNAME__ macro, [1-64](#)
user options, passing to main driver, [1-69](#)
user registers, [1-294](#)
USTAT registers, [1-297](#)
-U (undefine macro) compiler switch, [1-28](#), [1-65](#)

V

.VAR directive, declaring heap with, [1-284](#)
variable
 argument macros, [1-224](#)
 length array, [1-140](#), [1-224](#)
 statically initialized, [2-17](#)
variable expansion and MVE unroll, [2-71](#)
variable name length, [1-141](#)

VDK

project support selected, [1-65](#)
using thread-safe C/C++ run-time libraries with, [1-65](#)

vectorization

 annotations, [2-96](#)
 avoiding, [2-51](#)
 defined, [2-94](#)
 factor, [2-94](#)
 loop, [2-51](#), [2-62](#)
 transformation, [2-52](#)

-verbose (display command line) compiler switch, [1-66](#)

-version (display version) compiler switch, [1-66](#)

version information, displaying, [1-35](#)

__VERSION__ macro, [1-263](#)

__VERSIONNUM__ macro, [1-264](#)

virtual function lookup tables, [1-61](#), [1-137](#)

VisualDSP++

 debugger, [1-35](#)

 IDDE, [1-4](#)

 running compiler from command line, [1-3](#), [1-8](#)

__VISUALDSPVERSION__ macro, [1-264](#)

void functions (delay) examples, [1-333](#)

volatile

 C program constructs, [1-125](#)

 declarations, [2-5](#)

 register set, [1-180](#)

vtble section identifier, [1-61](#), [1-138](#)

vtbl section identifier, [1-61](#), [1-137](#), [1-138](#)

-v (version and verbose) compiler switch, [1-66](#)

W

warning keyword, [1-208](#)

- warning messages
 - control pragma, [1-207](#)
 - described, [2-5](#)
 - diagnostic, [2-5](#)
 - disabling, [2-5](#)
 - disabling all, [1-68](#)
 - errors, [1-67](#)
 - #warning directive, [1-145](#)
- Warn-protos (warn if incomplete prototype) compiler switch, [1-68](#)
- w (disable all warnings) compiler switch, [1-68](#), [2-6](#)
- Werror-limit (maximum compiler errors) compiler switch, [1-67](#)
- Werror-warnings compiler switch, [1-67](#)
- Wmis_suppress C compiler switch, [1-72](#)
- Wmis_warn rule_number C compiler switch, [1-73](#)
- W{...} number (override error message) compiler switch, [1-67](#), [2-5](#), [2-6](#)
- _WORD keyword, [1-159](#)
- workaround
 - rframe, [1-88](#)
 - swfa, [1-89](#)
 - __WORKAROUND__2126X_ANOMALY4__ macro, [1-88](#)
 - __WORKAROUND_2136X_ANOMALY2__ macro, [1-88](#)
 - __WORKAROUND_2136X_ANOMALY3__ macro, [1-88](#)
- workarounds
 - 21161-anomaly-45, [1-88](#)
 - 2126x-anomaly-4, [1-88](#)
 - 2136x-mem-write, [1-88](#)
 - 2136x-multi, [1-88](#)
 - all, [1-88](#)
 - anomalies, [1-88](#)
 - anomaly management, [1-85](#)
 - dag-stall, [1-88](#)
 - errata, [1-86](#)
 - interaction between -si-revision, -workaround and -no-workaround, [1-89](#)
 - not applied in asm(), [1-85](#)
 - no-workaround switch, [1-89](#)
 - rframe, [1-89](#)
 - valid workarounds list, [1-88](#)
 - workaround switch, [1-88](#)
- workarounds, not applied in asm(), [1-113](#)
- __WORKAROUNDS_ENABLED macro, [1-90](#), [1-264](#)
- workaround workaround_id compiler switch, [1-69](#), [1-88](#)
- wrapper project, [2-11](#)
- Wremarks (enable diagnostic warnings) compiler switch, [1-68](#), [2-6](#)
- write_extmem function, [1-247](#)
- write-files (enable driver I/O pipe) compiler switch, [1-69](#)
- write-opts (user options) compiler switch, [1-69](#)
- writes, array element, [2-33](#)
- Wterse (enable terse warnings) compiler switch, [1-68](#)

X

- .XML files, [1-41](#)
- xref (cross-reference list) compiler switch, [1-70](#)

INDEX

Z

ZERO_INIT qualifier, [1-204](#), [1-350](#)

ZERO_INIT section, [1-275](#)

zero length arrays, [1-224](#)